1991

An empirical study of fault detection by static units-consistency analysis

Browning, Judy A.

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/28161

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

AN EMPIRICAL STUDY OF FAULT DETECTION
BY STATIC UNITS-CONSISTENCY ANALYSIS

by

Judy A. Browning

September 1991

Thesis Advisor Timothy Shimeall

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY 2b. DCLASSIFICATION/DOWNGRADING SCHEDULE	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b. OFFICE SYMBOL (If Applicable) 37	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (city, state, and ZIP code) Monterey, CA 93943-5000	7b. ADDRESS (city, state, and ZIP code) Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 6b. OFFICE SYMBOL (If Applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (city, state, and ZIP code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. | PROJECT NO. | TASK NO. | WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
AN EMPIRICAL STUDY OF FAULT DETECTION BY STATIC UNIT-CONSISTENCY ANALYSIS (u)

12. PERSONAL AUTHOR(S)
Judy A. Browning

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (year, month,day)	15. PAGE COUNT
Master's Thesis	FROM TO	1991 September 18	92

16. SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17. COSATI CODES			18. SUBJECT TERMS (continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUBGROUP	software testing, static analysis, software tools, software experiments, fault detection, units-consistency analysis, dimensional analysis

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

With the increasing costs involved in software development, testing has become a more critical aspect of the software engineering process. Automatic methods, such as various static analysis techniques, may offer economic fault detection. This thesis analyzes a static analysis technique that allows users to associate units with variables in computer programs and to check that data transformations manipulate units in a consistent manner. A tool is designed and applied for this analysis. Its performance is measured by comparing the results with a previous study of other testing techniques in detecting faults. The results reveal that this technique consistently detected a narrow class of faults including some faults not found by other testing techniques. The results also show that application of this technique during the requirements and design phases of software development can identify faults associated with units-inconsistency early and reduce costs involved in developing a piece of software.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [X] UNCLASSIFIED/UNLIMITED [ ] SAME AS RPT. [ ] DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Timothy Shimeall	22b. TELEPHONE (Include Area Code) (408) 646-2509 22c. OFFICE SYMBOL CS/SM

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted — All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

An Empirical Study of Fault Detection by Static Units-Consistency Analysis

by

Judy A. Browning
Captain, United States Army
B.S., University of Southern Mississippi, 1985

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

September 1991

AN EMPIRICAL STUDY OF FAULT DETECTION BY STATIC UNITS-CONSISTENCY ANALYSIS

With the increasing costs involved in software development, testing has become a more critical aspect of the software engineering process. Automatic methods, such as various static analysis techniques, may offer economic fault detection. This thesis analyzes a static analysis technique that allows users to associate units with variables in computer programs and to check that data transformations manipulate units in a consistent manner. A tool is designed and applied for this analysis. Its performance is measured by comparing the results with a previous study of other testing techniques in detecting faults. The results reveal that this technique consistently detected a narrow class of faults including some faults not found by other testing techniques. The results also show that application of this technique during the requirements and design phases of software development can identify faults associated with units-inconsistency early and reduce costs involved in developing a piece of software.

TABLE OF CONTENTS

I. BACKGROUND AND PREVIOUS WORK

A. INTRODUCTION

With the proliferation of computers and the increasing costs involved in software development, testing has become a critical aspect of the software engineering process. Software testing usually requires more than half of the effort involved in producing working programs. Most programmers dislike testing. They have difficulty selecting data to detect faults in their programs. Fully automatic testing techniques help to find faults without forcing the programmers to select data.

The ANSI/IEEE Standard defines a fault as an accidental condition that causes a functional unit to fail to perform its required function. An error is a discrepancy between a computed, observed, or measured value or condition, and the true, specified, or theoretically correct value or condition. A failure is the inability of a system or system component to perform a required function within specified limits. (ANSI/IEEE, 1983, pp.18-19)

Techniques for detecting software faults are divided into two categories: dynamic analysis and static analysis. These techniques help identify different classes of faults within programs. The principle dynamic analysis technique is program testing (Howden, 1981, p. 210), which examines the behavior of a program during execution given certain sample test data. An example of this type of analysis is a test case analyzer that determines if all executable statements within a program are reached at least once.

Static analysis examines the source code and structure of programs for faults. This analysis occurs during the requirements, design, or implementation phases of the software development process. Static analysis involves examination of information in documents

created during these phases, but does not require actual execution of the program under development. It detects classes of faults that include uninitialized variables, undeclared variables, unreferenced variables, operand type mismatches, and conflicts between actual and formal parameters of modules. Static analysis techniques can detect different types of faults, but the use of these techniques may miss some important faults, specifically logical faults. There has been limited research examining the strengths and weaknesses of static analysis techniques. That research, however, has found several key limitations.

One limitation using static analysis techniques is the inability to track the value of a variable as it changes. An example of this is an array index. Since the value of array indexes is usually dynamically calculated there is no way for static analysis to identify the specific index value (Beizer, 1990, p.156). This forces static analysis techniques to treat a reference to any array element as a reference to all array elements. Static analysis techniques that are performed manually can be limiting because of the amount of detail involved. Since humans have difficulty in handling large amounts of details, manual techniques become less effective as program size increases.

Although static analysis techniques have limitations, their application may be beneficial under some conditions. Static analysis techniques may detect faults prior to execution of a program. If faults are detected earlier in the software development process, static analysis techniques may be more economical than other techniques. They may also be useful if the class of faults detected is not redundant to the faults detected by applying similar techniques. Static analysis may be used if it can be shown that the overall testing effort can be reduced. If testers are able to eliminate certain classes of faults early by using static analysis techniques or narrow the focus of the dynamic testing efforts then static analysis may be beneficial. Incorporating more effective static analysis techniques has been a continuing trend in language processor design.

Improving static analysis methods is the main purpose of this research. One way to improve static analysis is to add more application-based information into the analysis to help detect faults. Many programs in engineering and science deal with calculations involving physical units and units can be associated with variables in other application fields as well. Adding the capability to check units by static analysis requires defining units that are to be used, knowing the relationship among units, and defining the association of units with program variables. A static analysis technique that checks for consistency of units is different from other techniques that are built in to compilers such as type checking. With type checking, variables can be converted to other types by a syntactic means, but the unit associated with a variable is determined by known relationships or it is algebraically derived. Information associated with units may be derived from the application area and the program specification; the rest would need to be added by the analyst.

A computer program such as a language processor is a logical choice for this type of analysis because the technique involves repeated application of simple rules to easily-extracted program information. Manual checking of this sort is expensive, tedious and error-prone. A computer program would provide more consistent results with less cost. This check would not determine full program correctness, but rather, determine consistency in the program's data transformations that involve physical units.

B. RELATED RESEARCH

There has been some research in units-consistency checking that was concerned with implementations of units-consistency techniques. Karr did a study detailing how a programming language could keep track of physical units. He addressed the issues of what the set of elementary units would include and what would the relationship among those units be. He proposed that the user be allowed to reserve identifiers and declare relationships at their convenience. Once the user declared the units and their relationships,

a language compiler could then construct vectors representing relationships and apply linear algebra methods to check for unit consistency (Karr, 1978, pp. 385-391).

The research conducted by Bhargava focused on the dimensional aspects of units and their use in enhancing the reliability of mathematical modeling. He treated units as dimensions that represent non-numeric symbols. His method encoded units of measurement as prime numbers and manipulated the resulting expression numerically. Using the unique factorization theorem he developed a method to simplify dimensions, i.e., units, and verify dimensional equivalency. The primary focus of this research was its application in mathematical modeling systems, specifically the model validation and model solution phases (Bhargava, 1991, pp. 1-3).

Previous research outlined different methods for implementing units-consistency checking tools. Exploring dimensional simplification and verification of dimensional consistency of expressions in modeling formulation and model validation was the motivation for studies done by Bhargava (Bhargava, 1991, p. 3). Karr's research examined some language design considerations and compiler implementation issues (Karr, 1978, p. 386). Issues concerned with testing, such as increasing the reliability of programs, were left for future research. This thesis examines the question of reliability; it offers a first look at the relative effectiveness of a units-consistency analysis technique with other testing techniques.

C. RESEARCH QUESTIONS

This research involves the extension of a programming language with a construct for inclusion of physical units to allow for automatic detection of unit inconsistencies. The primary focus is a comparative analysis of this technique with other testing techniques applied in previous research. This information may affect the planning phase of software development and the selection of testing techniques.

One question addressed by this research deals with redundancy. If application of a units-consistency technique detects new faults, not previously revealed by other testing techniques, then it may be of value to testers. Applying non-redundant techniques helps to increase the reliability of a piece of software.

Another question addressed involves the cost of applying a units-consistency technique. Any software developer is concerned with costs. The costs associated with detection and correction of faults increase as each phase in the software development cycle is completed. If a technique can be applied in the earlier phases of software development, the costs associated with fault correction may be less.

A more informed software-development manager is able to make wiser decisions. This thesis provides information obtained from research that allows for better planning of the testing effort in the development process. It addresses the issue of whether units-consistency analysis meets criteria for effective use.

D. OVERVIEW

The first step in conducting this research was developing a tool to analyze a program for units-consistency. Chapter II gives a description of that tool. Chapter III is an analysis of the results of applying the tool, including a comparison of its performance with other testing techniques in detecting faults. The analysis in the chapter attempts to answer the research questions outlined in the previous section. The concluding chapter gives a summary of the significant results, suggestions to practitioners in the use of this technique and directions for future research.

II. RESEARCH DESCRIPTION

A. INTRODUCTION

The programming for this research project required building a static-analysis tool that checks consistency of physical units for each data transformation in a Pascal program (i.e., each assignment statement and parameter passing). The rationale behind checking only data transformations is that this is where values within programs frequently become contaminated.

A desire for flexibility and ease of testing led to structuring the tool into two steps. The first step is a program that parses the source code and generates an input file for the second step. The second step actually verifies the physical unit consistencies within a program by comparing the input file with a file that contains a list of valid relationships of variables and their physical units. This chapter describes the two programs that form the tool.

B. THE PARSER

The parser, called **Pparse**, takes as its input a Pascal program that uses an extended version of the basic Pascal grammar (Jenson and Wirth, 1974). The code for this parser is given in Appendix A. Two software-development tools, LEX and YACC (Mason and Brown, 1991), were used to develop **Pparse**. It is a basic Pascal parser with embedded C code that performs the appropriate semantic actions. **Pparse** follows a modified grammar that allows unit declarations following identifiers. Specialized comments, indicated by an ampersand immediately following the comment delimiter, form the unit declarations. The programmer can insert physical unit declarations after any identifier used in a program. This allows the user to associate units with constant declarations, type declarations, and

variable declarations at the beginning of the program, but has the added flexibility of placing unit declarations in data transformations.

The information generated by **Pparse** becomes the input file for the physical units checker. It provides an expression for each data transformation in the Pascal program being parsed. The curly brackets, "{}", delimit the beginning and end of each expression respectively. A line number appears after the open bracket that tells the user approximately where the data transformation occurs in the source code. The expression contains Pascal variables, which indicate array subscripts by the square brackets, “[]”, record references by a period “.”, and pointer references by a carrot “^”. Any combination of these aggregate data types can form complex variable references. For example, the reference "New^[][].BatDim.LinWid" is a variable that points to a two dimensional array containing the "BatDim" record with the subelement “LidWid”. The parser assigns units at the lowest level of reference, therefore whatever unit belongs to the subelement “LinWid” is the unit associated to this reference in a program. Because all elements of an array were assumed to have the same physical units, index expressions did not disambiguate units within an expression and were left out. **Pparse** describes other program references such as literal values using the notation “@@”. A reference to an literal string, character or number, appears as “@@unsigned_lit” and a reference to a set appears “@@set”.

Since many data transformations in programs involve assignment statements the expression format for the parser result file resembles a Pascal assignment statement. **Pparse** supports all Pascal operators. The parser lists the operator followed by a space and the number of arguments associated with it. The convention used for representing the arguments themselves lists the identifier (variable) followed by a space and the unit associated with it.

C. THE PHYSICAL UNITS CHECKER

This program called **unitcheck**, requires two ASCII input files, the parser results and a list of valid data transformations. Appendix B gives the code for **unitcheck**. A sample page from an output file produced by **unitcheck** is shown in Appendix D. The functional specification is the source of the valid data transformations. It is a database, of a sort, that identifies valid ways in which units associated with variables can be combined to form other units. This second input file is referred to as the **rulebase** for the testing tool. A sample page for the rulebase is shown in Appendix C. The diagram in Figure 1 gives a pictorial representation of the program.

The first thing **unitcheck** does is load the **rulebase** by calling the **load_list** function. **load_list** in turn uses a function called **read_exp** to load a parse tree for each individual data transformation in the **rulebase** file. As it builds each tree it returns a pointer to that tree and **load_list** adds the tree into a linked list. **load_list** returns the head of the list when it reaches the end of the **rulebase** file. This list represents valid data transformations derived from the functional specification. **unitcheck** then opens the parser result file and calls **read_exp** to load a parse tree for each data transformation. Each time **read_exp** loads a tree, **unitcheck** calls the **match** function to evaluate the validity of the units in the tree.

The **match** function has as its first parameter the head of the linked list created from **load_list** and its second parameter is the parse tree returned from the second call to **read_exp**. The **match** function immediately calls three functions that perform quick checks on the parse tree. **unitcheck** uses these functions to validate parse trees that do not require comparisons with the entire rulebase. The first function, **fast_compare1** validates data transformations involving assignments of a variable to another variable with the identical units or an assignment of an unitless literal to a variable. **fast_compare2**

traverses the parse tree and eliminates it if there are no units of measurement associated with any of its identifiers. The last non-rulebase check, **fast_compare3**, validates a parse tree if all of the identifiers within it have identical physical units and all the operators are unit-preserving. It accomplishes this by calling a recursive function, **traverse** that traverses the parse tree and based on each operator encountered, decides if the units are consistent. If none of the quick checks validate the expression then the **match** function calls **compare_exp** to validate the expression against the **rulebase**.

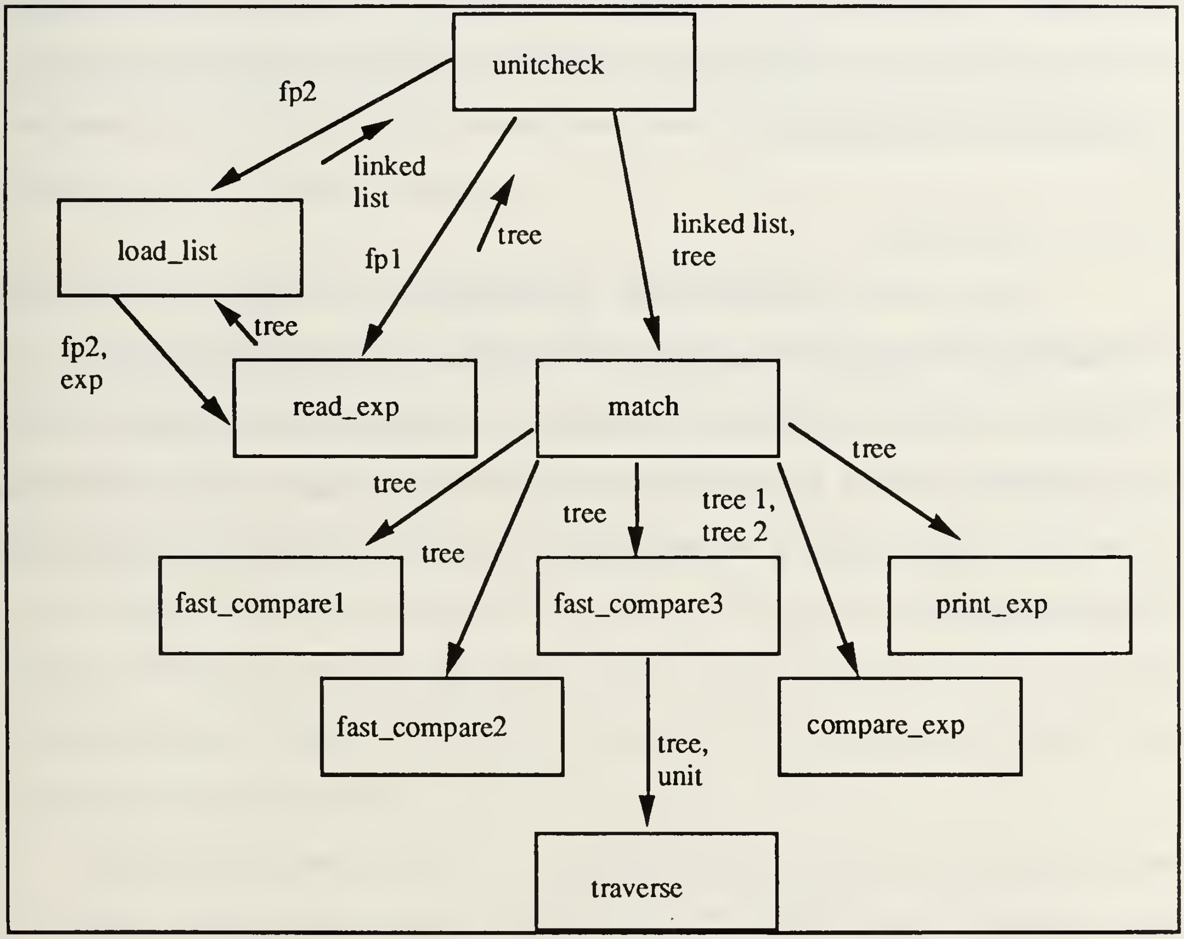

Figure 1: Structure Chart for Unitcheck

The **compare_exp**, also a recursive function, takes as its parameters a parse tree from the **rulebase** and the parse tree from the parser result file. It traverses both parse trees and verifies that the nodes in each of the trees are identical by comparing the identifier name and unit name at each node. If **compare_exp** finds two identical parse trees, it signals the **match** routine that in turn prints a statement to the standard output denoting the units in the expression are valid. If **compare_exp** traverses the two parse trees and finds the structures are different or there exists any inconsistent units, it notifies the **match** routine. If the **match** routine continues through the entire rulebase and does not find two identical expressions, then it prints a statement denoting the expression from the parser result file has invalid units.

D. SUMMARY

Using **Pparse** and **unitcheck**, a programmer can evaluate the variable data transformations within a Pascal program. The rulebase for this evaluation may derive from the rules of algebra, the program specification or hand-validated expressions from the program source code. To measure the utility of this type of tool in practice, it was applied to a set of eight versions of a Pascal program. The next chapter discusses that application and its results.

III. RESULTS

A. INTRODUCTION

In the past there has been a lack of empircal research dealing with dimensional analysis and units consistency. Furthermore, research that has been done was not of a comparitive nature. This experiment involved building a static units-consistency analysis tool to detect faults within programs and comparing the results with faults previously detected by other testing techniques. Section B describes the program versions used in the experiment. Section C describes the experiment procedure. The results of this research are provided and summarized in Section D.

B. DESCRIPTION OF EXPERIMENTAL SUBJECT

This research used a set of eight program versions written from a single specification for a combat simulation problem. An industrial specification obtained from TRW (Dobieski, 1979) provided the base for the specification of a combat simulation called CONFLICT (Shimeall, 1990). The specification is structured as a series of transformations to convert input data to an internal state vector, use that state vector to model combat, and transform the state vector to report combat results. It was the implementation of that series of transformations that the unit-consistency analysis tool evaluated in this experiment.

Upper-division computer science students performed all design and implementation activities on the program versions of the CONFLICT Specification. At the time these students were in a senior-level class on advanced software engineering methods. Two students worked as a team to write each version. Each of the teams worked separately of each other with minimal sharing of information between teams. The program versions

were developed to the point where unit testing would normally begin. Table 4.1 provides information about the individual program versions.

TABLE 4.1: CONFLICT VERSION SOURCE PROFILE FROM (Shimeall and Leveson, 1991, p. 178)

#	Modules	Version Source Lines	Code Lines
1	72	7503	2414
2	56	3452	1540
3	41	1480	1201
4	57	3663	2003
5	28	1834	1544
6	72	3065	2206
7	75	2734	1978
8	57	1896	1331

A disjoint set of students detected faults in the programs using five different testing techniques: code reading by stepwise abstraction, static data-flow analysis, run-time assertions inserted by the development participants, multi-version voting, and functional testing with follow-on structural testing. An administrator acted as final arbiter and decided which reports were faults and which were false alarms (Shimeall and Leveson, 1991, p. 175). Table 4.2 in section D gives a total of these previously detected faults.

There are two factors that make the program versions of the CONFLICT Specification suitable to study unit analysis. First, the assignment of physical units of measurements to the variables in the CONFLICT Specification was straightforward, as the variables model the physical world. Second, and most important, previous research on the program versions provided the basis for comparison with other fault detection techniques, specifically those studied in the previous experiment.

C. EXPERIMENT PROCEDURE

The purpose of this experiment was to develop a testing tool that detects faults resulting from unit inconsistencies within programs and compare the results with a previous study that disovered faults by different testing procedures. The experiment was conducted in a series of steps, the first being the assignment of physical units to the program versions. The next step was building up the rulebase. Once the rulebase was built the unit-checking tool was executed on each program version. A unit clash occurred when the axioms for algebraic manipulation of variables were followed and the physical unit in the left-hand side of an equation did not match the units derived from the right-hand side of the equation. The last step was the analysis of the output to determine if each unit clash was a fault. At each step, reviews were done that validated experiment procedures and data. The remainder of this section details each of the steps taken in the experiment.

The first step in the experiment was assigning physical units to global variables within the CONFLICT Specification. The decision of what kind of unit to assign to each variable was made based on the the specification itself. The system of measurement for the unit assignments was arbitrarily decided to be metric. Once units were assigned to all global variables, each program version was edited assigning units to appropriate parameters, function-return declarations and local variables. Units were assigned to parameters in the procedure or function declaration by adding them in the comment form discussed in Chapter II. Function-return declarations were assigned units by appending the function call with the appropriate unit, again in the comment form described in Chapter II. Since some functions performed services of a general nature, such as returning a minimum value, it was not possible to assign units to them. If a local variable was not associated with a global variable within the CONFLICT Specification, it was not assigned a unit.

The unitcheck program was then used to check this program version with the unitless-expression filtering mechnism disabled. Disabling this mechanism helped ensure that all variables were properly assigned units. These initial results were then reviewed for one more check to determine if units had been accidently left out or if any incorrect units had been added. After the review, any needed modifications were made to the unit associations in the source code of the program version.

Another problem that was addressed during the above process was how to build the rulebase. Since the output of the **unitcheck** procedure was a list of data transformations for the program version, a logical technique to build the rulebase was to evaluate each of the transformations in the output and determine if it should be added to the rulebase. The criteria for adding rules were: is it a valid rule and is it a rule that is likely to occur in other versions. The CONFLICT Specification provided the functional requirements that described variables and established the relationships between these variables. This was the basis for the development of the rulebase. If the rule was in the CONFLICT Specification it was then checked to ensure that algebraic manipulations followed appropriate composition and cancellation rules. If a rule was not general enough to be likely to appear in other program versions, it was left out of the rulebase. The rules that met the criteria were added to the rulebase against which the other program versions would be checked.

Once a rulebase was established, a fully-filtered unit check was done. The result was again evaluated manually, determining if there were any unit clashes. As unit clashes were detected they were noted for further examination.

When all the program versions were checked, the unit clashes were evaluated to ascertain if each unit clash could cause the program to fail. If a valid condition produced a program failure from the clash then it was classified as a fault, otherwise it was left as a unit clash. The reason for each clash was noted, for example, a clash that occured because

of the implementation of a particular version was classified as a coding unit clash. A final review of each unit clash was done comparing them with previously detected faults that were listed in the study by Shimeall and Leveson (Shimeall and Leveson, 1991, p 178). This last review helped establish the validity of the research results.

D. ANALYSIS OF RESULTS

Three general categories of questions guided the analysis of this data. The first category is did this testing technique reveal any faults that were not previously discovered by the other testing techniques. The second category is what are the strengths and weaknesses of the physical units checking tool. The final category is how can the tool be improved or better used. These questions are discussed in the next three sections.

1. Faults Detected by the Tool

Table 4.2 is a summation of research results after conducting the test for physical units consistency. The first two rows in Table 4.2 marked 'Total Transformations' and 'Filtered Transformation', reveal that a large number of data transformations were filtered out automatically. The parser generated the number of data transformations shown in the first row. The **unitcheck** program then filtered out many data transformations and reported the numbers in second row for further analysis. As these first two rows in Table 4.2 show, the **unitcheck** procedure filtered between 66 to 90 percent of the initial data transformations.

The row marked 'Coding Unit Clashes' in Table 4.2 shows that the unit clashes in the program versions of the CONFLICT Specification were usually the result of reuse of variables in a different context or module. Most of these clashes were not classified as faults within the program. However, there was one unit clash included that was found in each program version and classified as a fault. This clash is discussed in more detail later.

The next row marked 'Specification Clashes' revealed there were unit clashes that occurred because of inconsistencies within the CONFLICT Specification. These clashes were not a surprise since the specification was not previously analyzed for physical units consistency. The row marked 'Total Clashes' is the sum of the previous two lines and gives us the figure for all unit clashes detected by our research including all unit clashes that could result in a fault .

TABLE 4.2: RESULTS OF PHYSICAL UNITS-CONSISTENCY ANALYSIS

	1	2	3	4	5	6	7	8
Total Transformations	943	544	614	853	501	722	665	654
Filtered Transformations	96	128	64	131	148	246	222	167
Coding Unit Clashes	2	3	2	3	3	3	2	3
Specification Clashes	0	0	0	0	1	0	1	0
Total Clashes	2	3	2	3	4	3	3	3
Previous Known Faults	26	30	46	36	40	22	31	45
Previous Known Faults Revealed by Clashes	1	0	1	0	0	0	1	0
New Faults Revealed by Clashes	0	1	0	1	1	1	0	1

The first question answered by this data deals with the consistency of this technique in detecting faults. The data shown in the rows marked 'Previous Known Faults Revealed by Clashes' and 'New Faults Revealed by Clashes' in Table 4.2 show the consistent detection of faults in each of the program versions. The class of faults detected was quite narrow, relating to only one aspect of the specification. Further analysis revealed that this fault class did indeed result from use of analagous variables and units in each program version. It involved the numeric precision of a calculation specifically where the sum of *n* copies of term may not be equal to the product of multiplying that same term by *n*. This inequality results from the rounding mechanisms used in digital machines. An

inconsistent detection technique, multi-version voting, detected these faults in only three of the eight program versions.

With this in mind, the next question is what will this technique consistently not reveal about faults within the program versions. In many cases its just as important to know which kinds of faults are not revealed by using a tool. This gives testers an idea what techniques should be used in combinations so that deficiencies of one technique are compensated for by using another technique for fault detection. This tool reported only unit clashes occuring within data transformations. The row marked, 'Previously Known Faults' provides the results from the previous testing research that gives a total of 276 faults that were previously revealed. Further analysis in this area found that 210 out of those 276 involved no data transformations and hence would not be detected by the tool. Out of the 66 left, **unitcheck** filtered out 52 faults due to data transformations that did not involve a change of units. The filtering procedures were described in Chapter II and involved transformations that contain unitless variables, transformations where an unsigned literal is added to or subtracted from a variable, or transformations that contained identical units. Of the 14 remaining known faults, 11 coincidentally involved legal unit conversions.

The three remaining data transformations that were reported by **unitcheck** fell into the category of previously detected faults. The tool detected these faults in three different program versions of the CONFLICT Specification. These figures are shown in the row marked 'Previously Known Faults Revealed by Clashes'. All of the previously detected faults were revealed by multi-version voting. The procedure used in the previous experiment for this testing method compared the results of three program versions, looking for disagreement. This result naturally leads to the question of what gains there could be if this testing tool were used in conjunction with other types of testing methods. The rows

marked 'Previously Known Faults Revealed by Clashes' and 'New Faults Revealed by Clash' show that the units-consistency checking tool detected the fault consistently in all program versions. The previous research testing technique of multi-version voting detected these faults in only three program versions. These results show that this tool can be used with other testing techniques and detect faults that are not redundant. Although the units-consistency analysis revealed only a very restricted class of faults, it did in five out of eight program versions detect a fault that had not been detected by other testing techniques that used over 10,000 executions.

2. Strengths and Weaknesses of the Tool

A primary question in the minds of software development managers is the cost involved with a testing tool. They are concerned with the cost of the tool itself, whether its developed in house or purchased separately, and the cost of training testers to use the tool effectively. An advantage of this tool is that it could be developed in house for imperative languages by developing a parser or modifiying an existing one. The primary cost here is in time, but even there the cost is minimum. A experienced tester could implement a units-consistency checker in less than two months of full time work. As far as the cost of administering the test to the individual programs this would depend on the length of the program versions. For a program size of around 2000 lines of code it would take approximately three to five days of full-time work if the programmer had to actually analyze the requirements document for unit assignment, assign units in program versions, run the test, and validate results. This time could also be reduced if units analysis were introduced in the requirements and design phases because the tester would not have to perform the steps of analyzing the requirements documents for unit assignment and assigning units in the programs themselves. This use of the technique is described in greater detail below.

Another question that concerns program developers is are there aspects of this testing technique that will determine whether it is useful or not to a particular problem area? Scientific and engineering applications have long been able to use the technique of units analysis for faster and more accurate progam development. It could also be very useful in command and control applications where physical objects are manipulated. And lastly, integrating units into business applications may prove to be very beneficial. There is no limit for use of this technique in business applications because as long as units are assigned consistently and the axioms of algebraic manipulation of variables are applied, a units-consistency checking technique is applicable.

One last question that might concern a software development manager is how likely are the testers to be mislead by using this technique? This testing technique detects only a very narrow class of faults, specifically those that occur due to units that are inconsistent within data transformations. Testers should be aware that when this technique is applied, faults are revealed and can be eliminated, but this is a very restrictive set of faults and their removal does not certify program correctness.

These questions address some issues that software development managers might be concerned with, but as managers they are also looking for ways to better utilize any tools and resources available to them. Questions that address these issues are discussed in the following section.

3. Improvement and Other Uses of the Tool

The question of how to better utilize resources is always of importance to any manager regardless of what they are managing. The question with a technique such as units-consistency checking is how can it be used to increase the reliability of software and be more cost effective? The row in Table 4.2 marked ‘Specifiation Clashes’ revealed physical units clashes within the CONFLICT Specification itself. This data indicates that a

technique of this sort could be beneficial when used to detect unit inconsistencies within requirement and design documents. Reliability of software has become increasingly more important and CASE tools that check consistencies in the earlier phases of development are becoming more widespread. The capability for checking of units-consistency could be added to these CASE tools. Howden outlines a formal system that checks for consistencies within requirements and design documents in his work (Howden, 1981, pp. 103-105). The advantage of this is costs associated with detecting faults early on in the development cycle are much less than when faults are detected later.

Another improvement to the units-consistency checking technique lies in the interface analysis area. Interface analysis involves checking formal and actual parameters for consistency. This technique has generally been done by a language processor. The current version of the parser for units-consistency checking tool generated a data transformation for each parameter being passed. It then checked the units-consistency between formal and actual parameters. There were problems however because in some cases comparisons were made of variables when they were of different unit types. An example in the program versions of this experiment occured with a function that compared two values and returned the variable with the minimum value. The tool detected no fault when two variables with two different unit types were compared. The occurence of this problem could be avoided by making improvements to the parser that enabled it to better check parameters. All that is required for this type of interface analysis is a symbol table and rules for judging consistency. Since a symbol table is inherent to a parser, the only addition would be the rules for checking parameter consistency.

E. CONCLUSION

This chapter described the methodology used in conducting this research, as well as the results of the experiment. A question concerning the use of units analysis technique is

where is time likely to be lost. The process of using the tool in the experiment was iterative in nature, meaning that the same steps were followed for each program version that was tested. Some of these steps, such as assigning units within the requirements specification and program versions, could be eliminated if units-consistency techniques are applied earlier in the software development process. However, the process itself has to be somewhat iterative to avoid mistakes on the part of the tester. Unit inconsistencies that are reported at the various steps within the process should be compared with previous results to prevent errors and validate results.

Some care should be taken when using the results of this experiment. Multiple examinations by different individuals were conducted to check the results, but the experiments was conducted by students, not professional programmers and testers. Just one application was examined in this research and extensibility to other applications has yet to be determined. Statistical significance of results was not addressed due to the lack of population information about the number of faults occuring in programs or the class of faults that appear most often in programs. In general there is a lack of historical data and applicable theory in the study of faults that occur in programs and therefore statistical significance was not able to be addressed.

The next chapter gives a summary of the significant results of this research. It offers conclusions and recommendations to practioners concerning the results of this research and finally discusses directions for future work.

IV. CONCLUSION

A. INTRODUCTION

The primary purpose of this research was to develop and apply a tool for units-consistency checking and compare the results of using this tecnique with the performance of other testing techniques in detecting faults in computer programs. This chapter summarizes the significant results of this research in Section B. Section C offers recommendations to practioners in applying techniques of this sort. Section D concludes by giving suggestions for future research.

B. RESEARCH SUMMARY

The most significant result of this research is that it offers the first look at the relative effectiveness of a units-consistency checking technique, comparing it with other testing techniques. Although this technique deals with a very narrow class of fault detection, the results did reveal that it was able to detect new faults that had not been detected by applying other testing techniques and over 10,000 program executions. This technique was able to consistently detect this class of faults. This fact is significant because a units-consistency checking technique used in conjunction with other static analysis techniques may reduce the issues to be explored during the dynamic analysis testing phase.

The last notable outcome of the study of this technique is support for its application early in the software development cycle. If faults identified via unit inconsistencies are detected during the requirements and design phases of development, it is less costly to repair them than if they are detected in later phases of development.

C. RECOMMENDATIONS

The results of this research can help practioners in planning for software testing. Chapter I discussed the conditions when it may be beneficial to apply static analysis techniques. The first of those conditions was if faults were detected prior to execution of a program. The results show consistent detection of a narrow class of faults. The second condition was if the technique could detect faults early in the software development process. The unit clashes found in the CONFLICT Specification show that if a units-consistency technique is applied during the requirements and design phases of software development faults can be detected early. Adding units of measurements into the documentation during those phases of development would also help organize information about variables to be used in the implementation and testing phases and increase readability. The third condition was if the class of faults detected was not redundant. The results show that it is not redundant effort to apply this with other techniques. Eliminating the class of faults that are associated with units inconsistencies addresses the fourth condition, reducing the overall testing effort. Of course if the technique for units-consistencies is incorporated into CASE tools that can be used during the requirements and design phases, checking for units-consistency will become much easier. Based on this data, there is no reason to reject use of units-consistency analysis.

This technique should be applied in conjunction with other testing techniques simply because of the narrow class of faults that it is capable of detecting. The experience described in the previous chapter suggests areas where caution is needed in applying this technique, particularly in determination of which reports are faults and which are false alarms.

D. FUTURE WORK

There are several avenues of research that can be examined in the future as follow-up studies to this work. The first question that could be answered is can the idea of consistency be extended to include more general relationships than physical units of measurement, thus broadening the class of faults that can be detected. The use of units in this research allows for distinction between values. At a qualitative level there are values associated with program variables that need to be treated differently, e.g., a null pointer as opposed to all other pointer values. Detection of these values for variables could be included and would broaden the class of faults detected.

Another potential area of research might involve checking for units-consistency beyond a single data transformation. Conditional statements, blocks of assignment statements, and consistency between modules are all areas where there is room for units-consistency checking. Checking for units-consistency in these cases becomes increasingly more difficult because the analysis is taken away from a specific location and has to address under which paths and conditions would units remain consistent. One possible benefit in exploring this approach further is that information can be obtained about declarations and references of variables.

The potential for using a unit analysis technique in conjunction with CASE tools has been mentioned in several sections of this work. This use could prove to be the most cost effective in software development because inconsistencies, therefore potential faults, are eliminated early in the development process.

Further work can be done to examine how information involving units can be maintained in very large software projects. Questions that should be addressed include how should the data be divided, what kind of database structure is most advantageous for this type of information and would lend itself to updates made during the course of

development. It is critical for very large software projects that information concerning units and variables be accessable to multiple users. Additionally, these users may have little contact with each other over the course of development. Research in this area could help solve problems involving accessability as well as issues involving accuracy and timeliness of data inherent in large software projects.

Finally, further comparisons can be done to statistically establish the limitations and gains of using a units-consistency analysis technique. The analysis of faults that occur in software is a relatively new field of study and data dealing with this class of fault detection needs to be explored more throroughly. This research has established some initial observations, but many more questions are left unanswered.

APPENDIX A

PASCAL PARSER

```
/*************************************************************************
*
                    Yacc specification for Pascal
Original specification taken from:
      yacc grammar for Pascal based on ISO standard
      Compiler Design and Construction: Tools and Techniques
      Arthur B. Pyster
      Van Nostrand Reinhold Company
      Copyright 1988
      ISBM 0-442-27536-6
      pp 159-163
*************************************************************************
/
/* Data structure for tokens */
%union {
        int             t_int;
        char            *t_str;
        struct tnode    *t_node;
        char            *idlist[256];
       }
/***********     Declaration of all token types     ***********/
%token  <t_int> TOK_AMPERSAND
%token      <t_int>     TOK_AND
%token      <t_int>     TOK_ARRAY
%token      <t_int>     TOK_ASSIGN
%token      <t_int>     TOK_BEGIN
%token      <t_int>     TOK_CASE
%token      <t_int>     TOK_CLOSEBRACKET
%token      <t_int>     TOK_CLOSEPAREN
%token      <t_int>     TOK_COLON
%token      <t_int>     TOK_COMMA
%token      <t_int>     TOK_COMMENT1_START
%token      <t_int>     TOK_COMMENT1_END
%token      <t_int>     TOK_COMMENT2_START
%token      <t_int>     TOK_COMMENT2_END
%token      <t_int>     TOK_CONST
%token      <t_int>     TOK_DIV
%token      <t_int>     TOK_DIVIDE
%token      <t_int>     TOK_DO
%token      <t_int>     TOK_DOTDOT
%token      <t_int>     TOK_DOWNTO
%token      <t_int>     TOK_ELSE
%token      <t_int>     TOK_END
%token      <t_int>     TOK_EQUAL
%token      <t_int>     TOK_FILE
%token      <t_int>     TOK_FOR
%token      <t_int>     TOK_FORWARD
```

```
%token      <t_int>     TOK_FUNCTION
%token      <t_int>     TOK_GOTO
%token      <t_int>     TOK_GREATERTHAN
%token      <t_int>     TOK_GREATERTHANOREQUALTO
%token      <t_int>     TOK_IDENTIFIER
%token      <t_int>     TOK_IF
%token      <t_int>     TOK_IN
%token      <t_int>     TOK_LABEL
%token      <t_int>     TOK_LESSTHAN
%token      <t_int>     TOK_LESSTHANOREQUALTO
%token      <t_int>     TOK_MINUS
%token      <t_int>     TOK_MOD
%token      <t_int>     TOK_MULT
%token      <t_int>     TOK_NEWLINE
%token      <t_int>     TOK_NIL
%token      <t_int>     TOK_NOT
%token      <t_int>     TOK_NOTEQUAL
%token      <t_int>     TOK_OF
%token      <t_int>     TOK_OPENBRACKET
%token      <t_int>     TOK_OPENPAREN
%token      <t_int>     TOK_OR
%token      <t_int>     TOK_PACKED
%token      <t_int>     TOK_PERIOD
%token      <t_int>     TOK_PLUS
%token      <t_int>     TOK_POINTER
%token      <t_int>     TOK_PROCEDURE
%token      <t_int>     TOK_PROGRAM
%token      <t_int>     TOK_RECORD
%token      <t_int>     TOK_REPEAT
%token      <t_int>     TOK_SEMICOLON
%token      <t_int>     TOK_SET
%token      <t_int>     TOK_STRING
%token      <t_int>     TOK_THEN
%token      <t_int>     TOK_TO
%token      <t_int>     TOK_TYPE
%token      <t_int>     TOK_UNIT1
%token      <t_int>     TOK_UNIT2
%token      <t_int>     TOK_UNKNOWN
%token      <t_int>     TOK_UNSIGNED_INTEGER
%token      <t_int>     TOK_UNSIGNED_REAL
%token      <t_int>     TOK_UNTIL
%token      <t_int>     TOK_VAR
%token      <t_int>     TOK_WHILE
%token      <t_int>      TOK_WHITESPACE
%token      <t_int>     TOK_WITH
/* Standard or Pre-Defined Identifiers */
%token      <t_int>     TOK_BOOLEAN
%token      <t_int>     TOK_REAL
%token      <t_int>     TOK_INTEGER
%token      <t_int>     TOK_CHAR
/* Standard Procedures and Functions */
%token      <t_int>     TOK_ABS
%token      <t_int>     TOK_ARCTAN
%token      <t_int>     TOK_ARGC
%token      <t_int>     TOK_ARGV
```

```
%token      <t_int>     TOK_CARD
%token      <t_int>     TOK_CHR
%token      <t_int>     TOK_CLOCK
%token      <t_int>     TOK_COS
%token      <t_int>     TOK_DATE
%token      <t_int>     TOK_DISPOSE
%token      <t_int>     TOK_EOF
%token      <t_int>     TOK_EOLN
%token      <t_int>     TOK_EXP
%token      <t_int>     TOK_EXPO
%token      <t_int>     TOK_FLUSH
%token      <t_int>     TOK_GET
%token      <t_int>     TOK_HALT
%token      <t_int>     TOK_LINELIMIT
%token      <t_int>     TOK_LN
%token      <t_int>     TOK_MESSAGE
%token      <t_int>     TOK_NEW
%token      <t_int>     TOK_NULL
%token      <t_int>     TOK_ODD
%token      <t_int>     TOK_ORD
%token      <t_int>     TOK_PACK
%token      <t_int>     TOK_PAGE
%token      <t_int>     TOK_PRED
%token      <t_int>     TOK_PUT
%token      <t_int>     TOK_RANDOM
%token      <t_int>     TOK_READ
%token      <t_int>     TOK_READLN
%token      <t_int>     TOK_REMOVE
%token      <t_int>     TOK_RESET
%token      <t_int>     TOK_REWRITE
%token      <t_int>     TOK_ROUND
%token      <t_int>     TOK_SEED
%token      <t_int>     TOK_SIN
%token      <t_int>     TOK_SQR
%token      <t_int>     TOK_SQRT
%token      <t_int>     TOK_STLIMIT
%token      <t_int>     TOK_SUCC
%token      <t_int>     TOK_SYSCLOCK
%token      <t_int>     TOK_TEXT
%token      <t_int>     TOK_TIME
%token      <t_int>     TOK_TRUNC
%token      <t_int>     TOK_UNDEFINED
%token      <t_int>     TOK_UNPACK
%token      <t_int>     TOK_WALLCLOCK
%token      <t_int>     TOK_WRITE
%token      <t_int>     TOK_WRITELN
/* Precedence and Associativity among operators */
%left TOK_EQUAL  TOK_LESSTHAN  TOK_GREATERTHAN  TOK_NOTEQUAL
TOK_LESSTHANOREQUALTO TOK_GREATERTHANOREQUALTO TOK_IN
%left  TOK_PLUS TOK_MINUS TOK_OR
%left  TOK_MULT TOK_DIVIDE TOK_DIV TOK_AND TOK_MOD
%right TOK_NOT
%left  TOK_PERIOD
%right TOK_ELSE TOK_THEN
%right UNARY
```

```
/* declare non-terminal types */
%type  <t_str>  variable_trailers
%type  <t_str>  variable_trailer_func_parm_list
%type  <t_str>  variable
%type  <t_str>  identifier
%type  <t_str>  relational_op
%type  <t_str>  add_op
%type  <t_str>  mult_op
%type  <t_str>  unary_op
%type  <t_str>  unit_decl
%type  <t_str>  formal_parms_trailer
%type  <t_node> factor
%type  <t_node> signed_factor
%type  <t_node> expression
%type  <t_node> simple_expression
%type  <t_node> term
%type  <t_node> id_list
%type  <t_node> type
%type  <t_node> packable_type
%type  <t_node> ordinal_type
%type  <t_node> field_list
%type  <t_node> var_field_list
%type  <t_node> const_field_list
%type  <t_node> tag
%type  <t_node> cases
%type  <t_node> cases_trailer
%type  <t_node> opt_formal_parm_list
%type  <t_node> formal_parms
%type  <t_node> opt_return
/****************************************      YACC Specification
****************************************/
%%
program:
            TOK_PROGRAM identifier unit_decl TOK_OPENPAREN
                  io_list TOK_CLOSEPAREN TOK_SEMICOLON
            {
             TOS = 0;
             for (i = 0;i < MAX_TABSIZE; i++)
             strcpy(symtab[TOS][i].nodename,EMPTY);
               /* initializes symbol table */
             templ=(treenode *)CALLOC(1,sizeof(treenode),"program");
             strcpy(templ->nodename, $2);
             strcpy(templ->unitname, $3);
             add_sym(templ,symtab[TOS]);
             for (i = 0; i < MAX_TABSIZE; i++)
             strcpy(typetab[i].nodename,EMPTY);
               /*initializes type table */
             }
            block TOK_PERIOD
      ;
io_list :
            id_list
      |
      ;
id_list :
```

```
            identifier unit_decl
                {
             temp1=(treenode *)CALLOC(1,sizeof(treenode),
               "id_list1");
                 strcpy(temp1->nodename, $1);
                 strcpy(temp1->unitname, $2);
                 $$ = temp1;
                }
      |     identifier unit_decl TOK_COMMA id_list
                { temp1=(treenode *)CALLOC(1,sizeof(treenode),
               "id_list2");
                 strcpy(temp1->nodename, $1);
                 strcpy(temp1->unitname, $2);
                 temp1->rightchild = $4;
                 $$ = temp1;
                }
      ;
block :
            opt_labels opt_constants opt_types
            opt_variables opt_pf_heading_dcls
            TOK_BEGIN
            statements TOK_END
      ;
opt_labels :
            TOK_LABEL integer_list TOK_SEMICOLON
      |
      ;
integer_list :
            integer
      |     integer TOK_COMMA integer_list TOK_SEMICOLON
      ;
integer :
            TOK_UNSIGNED_INTEGER
      ;
opt_constants :
            TOK_CONST constant_dcls
      |
      ;
opt_types :
            TOK_TYPE
                  {
                   for (deftop=0; deftop<300; deftop++)
                           deferred[deftop]=NULL;
                   deftop=-1;
                  }
            type_dcls
                  {
                   for (;deftop>=0; deftop--) {
                     temp1 =
                           type_lookup(deferred[deftop]->leftchild-
>nodename);
                     if (temp1 != NULL) {
                         deferred[deftop]->leftchild = temp1;
                             deferred[deftop]->marked = 1;
                       break_link(temp1);
```

```
                                clear_mark(templ);
                                deferred[deftop]->marked = 0;
                           }
                        }
                  }

        |
        ;
opt_variables :
            TOK_VAR variable_dcls
        |
        ;
opt_pf_heading_dcls :
            opt_pf_heading_dcls procfunction_heading
              TOK_SEMICOLON block_directive TOK_SEMICOLON {TOS--;}
        |
        ;
block_directive :
            block
        |   directive
        ;
directive :
            TOK_FORWARD
        ;
statements :
            statement
        |   statements TOK_SEMICOLON statement
        ;
constant_dcls :
            identifier TOK_EQUAL constant unit_decl TOK_SEMICOLON
            {
             templ=(treenode *)CALLOC(1,sizeof(treenode),
               "constant_dcls");
             strcpy(templ->nodename, $1);
             strcpy(templ->unitname, $4);
             add_sym(templ,symtab[TOS]);
            }

        |   constant_dcls identifier TOK_EQUAL constant unit_decl
TOK_SEMICOLON
            {
                              t e m p l = ( t r e e n o d e
*)CALLOC(1,sizeof(treenode),"constant_dcls");
             strcpy(templ->nodename, $2);
             strcpy(templ->unitname, $5);
             add_sym(templ,symtab[TOS]);
            }
        ;
variable_dcls :
            id_list TOK_COLON type TOK_SEMICOLON
               {
            current = $1;
            while (current != NULL)
                  {
               next = current->rightchild;
```

```
                current->rightchild = NULL;
                    add_sym(current, symtab[TOS]);
                if ( (strcmp($3->nodename,"[]") != 0) &&
                     (strcmp($3->nodename,".") != 0) &&
                     (strcmp($3->nodename,"^") != 0) )
                  build_sym(current->nodename, $3->leftchild);
                else
                  build_sym(current->nodename,$3);
                current = next;
                   }
                 }
        |     variable_dcls id_list TOK_COLON type TOK_SEMICOLON
                  {current = $2;
              while (current != NULL)
                     {
                 next = current->rightchild;
                 current->rightchild = NULL;
                    add_sym(current, symtab[TOS]);
                if ( (strcmp($4->nodename,"[]") != 0) &&
                     (strcmp($4->nodename,".") != 0) &&
                     (strcmp($4->nodename,"^") != 0) )
                  build_sym(current->nodename, $4->leftchild);
                else
                  build_sym(current->nodename,$4);
                 current = next;
                   }
                 }
        ;
statement :
              opt_label unlabeled_statement
        ;
opt_label :
              TOK_UNSIGNED_INTEGER TOK_COLON
        |
        ;
unlabeled_statement :
              variable unit_decl TOK_ASSIGN expression
              {printf("{ %d\n",yylineno);
                  if (strcmp($2,EMPTY) != 0)
               printf(":= 2\n%s %s\n",$1,$2);
              else
              {
                 unit = lookup($1); /*lookup returns unit_name */
               printf(":= 2\n%s %s\n",$1,unit);
              }
                   rhs = print_node($4);
               if (rhs == FALSE) printf("something is wrong\n");
               printf("}\n");
                 }
        |     identifier
              {
              parmtop = 0;
              for (i = 0;i < MAXSIZE; i++)
                cur_parms[i] = NULL;
                 /* initializes parameter table */
```

```
        temp_id = parm_lookup($1);
        if (temp_id != NULL)
          temp_id = temp_id->rightchild;
        }
        opt_proc_parameter_list
|       TOK_BEGIN
        {
        parmtop = 0;
        for (i = 0;i < MAXSIZE; i++)
          cur_parms[i] = NULL;
           /* initializes parameter table */
        }
        statements TOK_END
|       TOK_IF
        {
        parmtop = 0;
        for (i = 0;i < MAXSIZE; i++)
          cur_parms[i] = NULL;
           /* initializes parameter table */
        }
        expression opt_else
|       TOK_WHILE
        {
        parmtop = 0;
        for (i = 0;i < MAXSIZE; i++)
          cur_parms[i] = NULL;
           /* initializes parameter table */
        }
        expression TOK_DO statement
|       TOK_CASE
        {
        parmtop = 0;
        for (i = 0;i < MAXSIZE; i++)
          cur_parms[i] = NULL;
           /* initializes parameter table */
        }
        expression TOK_OF case_body TOK_END
|       TOK_REPEAT
        {
        parmtop = 0;
        for (i = 0;i < MAXSIZE; i++)
          cur_parms[i] = NULL;
           /* initializes parameter table */
        }
        statements TOK_UNTIL expression
|       TOK_FOR
        {
        parmtop = 0;
        for (i = 0;i < MAXSIZE; i++)
          cur_parms[i] = NULL;
           /* initializes parameter table */
        }
        identifier TOK_ASSIGN expression
        direction
        {
```

```
            parmtop = 0;
            for (i = 0;i < MAXSIZE; i++)
              cur_parms[i] = NULL;
               /* initializes parameter table */
            }
            expression TOK_DO statement
      |     TOK_WITH variable_list TOK_DO statement
      |     TOK_GOTO TOK_UNSIGNED_INTEGER
      |
      ;
opt_else :  TOK_THEN statement TOK_ELSE statement
      |     TOK_THEN statement
      ;
variable_list :
            variable
      |     variable_list TOK_COMMA variable
      ;
constant_list :
            constant
      |     constant_list TOK_COMMA constant
      ;
case_body :
            constant_list TOK_COLON
            statement case_trailer
      ;
case_trailer :
            TOK_SEMICOLON
      |     TOK_SEMICOLON case_body
      |
      ;
direction :
            TOK_DOWNTO
      |     TOK_TO
      ;
opt_proc_parameter_list :
            TOK_OPENPAREN
                expression_opt_formats_list TOK_CLOSEPAREN
      |
      ;
expression_opt_formats_list :
            expression_opt_formats
      |     expression_opt_formats_list TOK_COMMA expression_opt_formats
      ;
expression_opt_formats :
            expression opt_formats
            {
             if (parmtop >= 0 && cur_parms[parmtop] != NULL)
             {
               printf("{ %d\n", yylineno);
               printf(":= 2\n%s %s\n",cur_parms[parmtop]->nodename,
                  cur_parms[parmtop]->unitname);
                   rhs = print_node($1);
               cur_parms[parmtop] = cur_parms[parmtop]->rightchild;
             }
                }
```

```
      ;
opt_formats :
            TOK_COLON expression
      |     TOK_COLON expression TOK_COLON expression
      |
      ;
expression_list :
            expression
            {
             if (parmtop >= 0 && cur_parms[parmtop] != NULL)
             {
               printf("{ %d\n",yylineno);
               printf(":= 2\n%s %s\n",cur_parms[parmtop]->nodename,
                  cur_parms[parmtop]->unitname);
                   rhs = print_node($1);
               cur_parms[parmtop] = cur_parms[parmtop]->rightchild;
               printf("}\n");
             }
                }
      |     expression_list TOK_COMMA expression
            {
             if (parmtop >= 0 && cur_parms[parmtop] != NULL)
             {
               printf("{ %d\n",yylineno);
               printf(":= 2\n%s %s\n",cur_parms[parmtop]->nodename,
                  cur_parms[parmtop]->unitname);
                   rhs = print_node($3);
               cur_parms[parmtop] = cur_parms[parmtop]->rightchild;
               printf("}\n");
             }
                }
      ;
expression :
            expression relational_op simple_expression
            { t e m p 1           =           ( t r e e n o d e
*)CALLOC(1,sizeof(treenode),"expression");
                 strcpy(templ->nodename, $2);
                 templ->leftchild = $1;
                 templ->rightchild = $3;
                 $$ = templ;
                }
      |     simple_expression  {$$ = $1;}
      ;
simple_expression :
            term {$$ = $1;}
      |     simple_expression add_op term
                {
                 templ = (treenode *)CALLOC(1,sizeof(treenode), "simple
exp");
                 strcpy(templ->nodename, $2);
                 templ->leftchild = $1;
                 templ->rightchild = $3;
                 $$ = templ;
                }
      ;
```

```
term :
            term mult_op signed_factor
                {
                 temp1 = (treenode *)CALLOC(1,sizeof(treenode),"term");
                 strcpy(temp1->nodename, $2);
                 temp1->leftchild = $1;
                 temp1->rightchild = $3;
                 $$ = temp1;
                }
      |     signed_factor {$$ = $1;}
      ;
signed_factor :
            unary_op factor
                {
                                                       temp1   =   (treenode
*)CALLOC(1,sizeof(treenode),"signed_factor");
                 strcpy(temp1->nodename, $1);
                 temp1->leftchild = NULL;
                 temp1->rightchild = $2;
                 $$ = temp1;
                }
      |     factor {$$ = $1;}
      ;
unary_op :
            unary_op
            { tmp=CALLOC((MAXSIZE),1,"unary op1");
              strncpy(tmp,yytext,MAXSIZE);
            }
            TOK_PLUS    %prec UNARY
            {$$ = tmp;}
      |     unary_op
            { tmp=CALLOC((MAXSIZE),1,"unary op2");
              strncpy(tmp,yytext,MAXSIZE);
            }
            TOK_MINUS   %prec UNARY
            {$$ = tmp;}
      |     unary_op
            { tmp=CALLOC((MAXSIZE),1,"unary op3");
              strncpy(tmp,yytext,MAXSIZE);
            }
            TOK_NOT     %prec UNARY
            {$$ = tmp;}
      |
            { tmp=CALLOC((MAXSIZE),1,"unary op4");
              strncpy(tmp,yytext,MAXSIZE);
            }
            TOK_PLUS    %prec UNARY
            {$$ = tmp;}
      |
            { tmp=CALLOC((MAXSIZE),1,"unary op5");
              strncpy(tmp,yytext,MAXSIZE);
            }
            TOK_MINUS   %prec UNARY
            {$$ = tmp;}
      |
```

```
            { tmp=CALLOC((MAXSIZE),1,"unary op6");
              strncpy(tmp,yytext,MAXSIZE);
            }
            TOK_NOT      %prec UNARY
            {$$ = tmp;}
      ;
factor :
            identifier
              {
               temp_id = parm_lookup($1);
               if (temp_id != NULL)
               {
                 parmtop++;
                 cur_parms[parmtop] = temp_id->rightchild;
               }
              }
            variable_trailer_func_parm_list unit_decl
              {
               templ = (treenode *)CALLOC(1,sizeof(treenode),
                  "factorl");
               strcpy(templ->nodename,$1);
                   strcat(templ->nodename,$3);
               strcpy(templ->unitname,$4);
                   if (strcmp(templ->unitname,EMPTY) == 0)
                   {
                    unit = lookup(templ->nodename); /*lookup returns
unit_name */
                     strcpy(templ->unitname,unit);
                   }
               $$ = templ;
              }
      |     TOK_OPENPAREN expression TOK_CLOSEPAREN {$$ = $2;}
      |     unsigned_literal unit_decl
            {
            templ = (treenode *)CALLOC(1,sizeof(treenode),"factor2");
            strcpy(templ->unitname,$2);
            strcpy(templ->unitname,"@@unsigned_lit");
            $$ = templ;
            }
      |     TOK_OPENBRACKET opt_elipsis_list TOK_CLOSEBRACKET
              unit_decl
            {
            templ = (treenode *)CALLOC(1,sizeof(treenode),
              "factor3");
            strcpy(templ->unitname,$4);
            strcpy(templ->unitname,"@@set");
            $$ = templ;
            }
      ;
unit_decl:
                TOK_UNIT1 identifier TOK_COMMENT1_END
                {$$ = $2;}
        |       TOK_UNIT2 identifier TOK_COMMENT2_END
                {$$ = $2;}
        |       {$$ = EMPTY;}
```

```
         ;
variable_trailer_func_parm_list :
                variable_trailers {$$ = $1;}
        |       TOK_OPENPAREN expression_list TOK_CLOSEPAREN
                {if (parmtop>0) parmtop--; $$ = EMPTY;}
        ;
opt_elipsis_list :
                elipsis_list
        |
        ;
elipsis_list :
                elipsis
        |       elipsis_list TOK_COMMA elipsis
        ;
elipsis :
                expression
        |       expression TOK_DOTDOT expression
        ;
relational_op :
                    { tmp=CALLOC((MAXSIZE),1,"relational op1");
                      strncpy(tmp,yytext,MAXSIZE);
                    }
                TOK_EQUAL {$$ = tmp;}
        |
                    { tmp=CALLOC((MAXSIZE),1,"relational op2");
                      strncpy(tmp,yytext,MAXSIZE);
                    }
                TOK_GREATERTHAN {$$ = tmp;}
        |
                    { tmp=CALLOC((MAXSIZE),1,"relational op3");
                      strncpy(tmp,yytext,MAXSIZE);
                    }
                TOK_GREATERTHANOREQUALTO {$$ = tmp;}
        |
                    { tmp=CALLOC((MAXSIZE),1,"relational op4");
                      strncpy(tmp,yytext,MAXSIZE);
                    }
                TOK_IN {$$ = tmp;}
        |
                    { tmp=CALLOC((MAXSIZE),1,"relational op5");
                      strncpy(tmp,yytext,MAXSIZE);
                    }
                TOK_LESSTHAN {$$ = tmp;}
        |
                    { tmp=CALLOC((MAXSIZE),1,"relational op6");
                      strncpy(tmp,yytext,MAXSIZE);
                    }
                TOK_LESSTHANOREQUALTO {$$ = tmp;}
        |
                    { tmp=CALLOC((MAXSIZE),1,"relational op7");
                      strncpy(tmp,yytext,MAXSIZE);
                    }
                TOK_NOTEQUAL {$$ = tmp;}
        ;
add_op :
```

```
            { tmp=CALLOC((MAXSIZE),1,"add op1");
              strncpy(tmp,yytext,MAXSIZE);
            }
        TOK_MINUS {$$ = tmp;}
    |
            { tmp=CALLOC((MAXSIZE),1,"add op2");
              strncpy(tmp,yytext,MAXSIZE);
            }
        TOK_OR {$$ = tmp;}
    |
            { tmp=CALLOC((MAXSIZE),1,"add op3");
              strncpy(tmp,yytext,MAXSIZE);
            }
        TOK_PLUS {$$ = tmp;}
    ;
mult_op :
            { tmp=CALLOC((MAXSIZE),1,"mult op1");
              strncpy(tmp,yytext,MAXSIZE);
            }
        TOK_AND {$$ = tmp;}
    |
            { tmp=CALLOC((MAXSIZE),1,"mult op2");
              strncpy(tmp,yytext,MAXSIZE);
            }
        TOK_DIV{$$ = tmp;}
    |
            { tmp=CALLOC((MAXSIZE),1,"mult op3");
              strncpy(tmp,yytext,MAXSIZE);
            }
        TOK_DIVIDE{$$ = tmp;}
    |
            { tmp=CALLOC((MAXSIZE),1, "mult op4");
              strncpy(tmp,yytext,MAXSIZE);
            }
        TOK_MOD{$$ = tmp;}
    |
            { tmp=CALLOC((MAXSIZE),1, "mult op5");
              strncpy(tmp,yytext,MAXSIZE);
            }
        TOK_MULT{$$ = tmp;}
    ;
variable :
        identifier variable_trailers
        {tmp = strcat($1,$2);
         $$ = tmp;
        }
    ;
variable_trailers :
        TOK_OPENBRACKET {parmtop++; cur_parms[parmtop] = NULL;}
            expression_list {parmtop--;}
            TOK_CLOSEBRACKET  variable_trailers
        {tmp1 = CALLOC((strlen($6)+3),1,"var trailer1");
             tmp1 = strcpy(tmp1,"[]");
         tmp1 = strcat(tmp1,$6);
             free($6);
```

```
        $$ = tmpl;
        }
   |    TOK_PERIOD identifier  variable_trailers
        {tmpl = CALLOC((strlen($3)+strlen($2)+2),1,"var trailer2");
            tmpl = strcpy(tmpl,".");
            tmpl = strcat(tmpl,$2);
            tmpl = strcat(tmpl,$3);
            free($2);
         free($3);
        $$ = tmpl;
        }
   |    TOK_POINTER  variable_trailers
        {tmpl = CALLOC((strlen($2)+2),1,"var trailer3");
            tmpl = strcpy(tmpl,"^");
            tmpl = strcat(tmpl,$2);
         free($2);
        $$ = tmpl;
            }
     |      {/* printf("Default rule\n"); */
            tmpl = CALLOC((1),1,"var trailer4");
         *tmpl='\0';
         $$ = tmpl;
        }
   ;
constant :
        TOK_PLUS unsigned_constant                          %prec UNARY
   |    TOK_MINUS unsigned_constant                         %prec UNARY
   |    unsigned_constant
   ;
unsigned_literal :
        TOK_UNSIGNED_REAL
   |    TOK_UNSIGNED_INTEGER
   |    TOK_STRING
   |    TOK_NIL
   ;
unsigned_constant :
        identifier
   |    unsigned_literal
   ;

type :
        TOK_POINTER unit_decl identifier
        {templ = (treenode *) CALLOC(1,sizeof(treenode),"type");
             strcpy(templ->nodename,"^");
             templ->leftchild = type_lookup($3);
             templ->rightchild = NULL;
             if (templ->leftchild == NULL){
              /* type not yet defined */
              templ->leftchild = (treenode *)
                    CALLOC(1,sizeof(treenode),"deferred type");
              strcpy(templ->leftchild->nodename,$3);
              deferred[++deftop] = templ;
          }
         strcpy(templ->unitname,$2);
             $$ = templ;
```

```
            }
      |     ordinal_type
            {$$ = $1;}
      |     opt_packed packable_type
            {$$ = $2;}
      ;
packable_type :
            TOK_ARRAY TOK_OPENBRACKET ordinal_type_list
              TOK_CLOSEBRACKET unit_decl TOK_OF type
            {
                                         t e m p 1 = ( t r e e n o d e
*)CALLOC(1,sizeof(treenode),"packable_type1");
              strcpy(temp1->nodename,"[]");
              strcpy(temp1->unitname,$5);
              temp1->leftchild = $7;
                      /* find out if leftchild is a named type, and
substitute */
                  /*   definition for type name */
                  if (strcmp(temp1->leftchild->nodename,"[]")!=0 &&
                      strcmp(temp1->leftchild->nodename,"^")!=0 &&
                  strcmp(temp1->leftchild->nodename,".")!=0 &&
                  temp1->leftchild->leftchild != NULL &&
                  temp1->leftchild->rightchild == NULL)
                     temp1->leftchild= temp1->leftchild->leftchild;
              $$ = temp1;
            }
      |     TOK_RECORD unit_decl field_list TOK_END
            {
                                         t e m p 1 = ( t r e e n o d e
*)CALLOC(1,sizeof(treenode),"packable_type2");
                  strcpy(temp1->nodename,".");
                  strcpy(temp1->unitname,$2);
                  temp1->leftchild = $3;
              $$ = temp1;
            }
      |     TOK_FILE unit_decl TOK_OF type
                {
                  temp1=(treenode
*)CALLOC(1,sizeof(treenode),"packable_type3");
                  strcpy(temp1->nodename,"^");
                  strcpy(temp1->unitname,$2);
                  temp1->leftchild = $4;
                  $$ = temp1;
            }
      |     TOK_SET unit_decl TOK_OF ordinal_type
                {
                  temp1=(treenode
*)CALLOC(1,sizeof(treenode),"packable_type4");
                  strcpy(temp1->nodename,EMPTY);
                  strcpy(temp1->unitname, $2);
                  $$ = temp1;
                }
      ;
ordinal_type_list :
            ordinal_type
```

```
        |     ordinal_type_list TOK_COMMA ordinal_type
        ;
ordinal_type :
              identifier unit_decl
              {
                                                   t e m p 1 = ( t r e e n o d e
*)CALLOC(1,sizeof(treenode),"ordinal_type1");
                                             if  ($1  ==  EMPTY)  strcpy(temp1-
>nodename,"@@predefined");
                else strcpy(temp1->nodename,$1);
                strcpy(temp1->unitname,$2);
                temp2 = type_lookup(temp1->nodename);
                if ($2 == EMPTY && temp2 != NULL)
                 /* unitname is empty, so copy it from typetab */
                {
                  strcpy(temp1->unitname, temp2->unitname);
                }
                temp1->leftchild = temp2;
                $$ = temp1;
              }
        |     TOK_OPENPAREN id_list TOK_CLOSEPAREN unit_decl
              {
                current = $2;
                while(current != NULL)
                {
                  strcpy(current->unitname,$4);
                      next = current->rightchild;
                      current->rightchild = NULL;
                  add_sym(current,symtab[TOS]);
                  current = next;
                }
                                                   t e m p 1 = ( t r e e n o d e
*)CALLOC(1,sizeof(treenode),"ordinal_type2");
                strcpy(temp1->nodename,"@@enumerated");
                strcpy(temp1->unitname,$4);
                $$ = temp1;
              }
        |     constant TOK_DOTDOT constant unit_decl
              {
                                                   t e m p 1 = ( t r e e n o d e
*)CALLOC(1,sizeof(treenode),"ordinal_type3");
                strcpy(temp1->nodename,"@@subrange");
                strcpy(temp1->unitname,$4);
                $$ = temp1;
              }
        ;
field_list : const_field_list TOK_SEMICOLON var_field_list
              {current = $1;
               if (current != NULL) {
                 while (current->rightchild != NULL)
                   current = current->rightchild;
                 current->rightchild = $3;
                 /* put var_field_list at end of const_field_list */
                 $$ = $1;
                 /* return head (const_field_list) of new list */
```

```
              }
             else $$ = $3;
             /* const_field_list is null, so return var_field_list as
new list*/
            }
      |     const_field_list {$$ = $1;}
      |     const_field_list TOK_SEMICOLON {$$ = $1;}
      |     var_field_list {$$ = $1;} ;
var_field_list: TOK_CASE tag TOK_OF cases
            {templ = $2;
                if (templ != NULL)
                  temp1->rightchild = $4;
                else templ = $4;
             $$ = templ;} ;
const_field_list:
            id_list TOK_COLON type
            {
              current = $1;
              while (current != NULL)
              {
                current->leftchild = $3;
                      /* find out if leftchild is a named type, and
substitute */
                  /*   definition for type name */
                  if (strcmp(current->leftchild->nodename,"[]")!=0 &&
                      strcmp(current->leftchild->nodename,"^")!=0 &&
                  strcmp(current->leftchild->nodename,".")!=0 &&
                  current->leftchild->leftchild != NULL &&
                  current->leftchild->rightchild == NULL)
                     current->leftchild= current->leftchild->leftchild;
                if (current->unitname == EMPTY)
                  strcpy(current->unitname,  current->leftchild-
>unitname);
                current = current->rightchild;
              }
              $$ = $1;
            }
      |     const_field_list TOK_SEMICOLON id_list TOK_COLON type
            {
              current = $3;
              while (current != NULL)
              {
                current->leftchild = $5;
                      /* find out if leftchild is a named type, and
substitute */
                  /*   definition for type name */
                  if (strcmp(current->leftchild->nodename,"[]")!=0 &&
                      strcmp(current->leftchild->nodename,"^")!=0 &&
                  strcmp(current->leftchild->nodename,".")!=0 &&
                  current->leftchild->leftchild != NULL &&
                  current->leftchild->rightchild == NULL)
                     current->leftchild= current->leftchild->leftchild;
                if (current->unitname == EMPTY)
                  strcpy(current->unitname,  current->leftchild-
>unitname);
```

```
                next = current;
                current = current->rightchild;
              }
              next->rightchild = $1;
              $$ = $3;
             }
        ;
tag :
            identifier {$$ = NULL;}                    /* really type
identifier */
        |   identifier unit_decl TOK_COLON type
            { templ=(treenode *)CALLOC(1,sizeof(treenode),"tag");
              strcpy(templ->nodename, $1);
              strcpy(templ->unitname, $2);
              templ->leftchild = $4;
                      /* find out if leftchild is a named type, and
substitute */
                  /*   definition for type name */
                  if (strcmp(templ->leftchild->nodename,"[]")!=0 &&
                      strcmp(templ->leftchild->nodename,"^")!=0 &&
                      strcmp(templ->leftchild->nodename,".")!=0 &&
                      templ->leftchild->leftchild != NULL &&
                      templ->leftchild->rightchild == NULL)
                     templ->leftchild = templ->leftchild->leftchild;
              $$ = templ;
            }
        ;
cases :
            constant_list TOK_COLON TOK_OPENPAREN field_list
              TOK_CLOSEPAREN cases_trailer
            {current = $4;
             if (current != NULL) {
               while (current->rightchild != NULL)
                 current = current->rightchild;
               current->rightchild = $6;
               /* put cases_trailer at end of field_list */
               $$ = $4;
               /* return head of new list */
               }
              else $$ = $6;
              /* field_list is null, so return cases_trailer as new list
*/
            }
        ;
cases_trailer :
            TOK_SEMICOLON cases
            {$$ = $2;}
        |   TOK_SEMICOLON
            {$$ = NULL;}
        |   {$$ = NULL;}
        ;
procfunction_heading :
            TOK_PROCEDURE identifier unit_decl opt_formal_parm_list
            {
                TOS++;
```

```
                for (i = 0;i < MAX_TABSIZE; i++)
                 strcpy(symtab[TOS][i].nodename,EMPTY);
                templ=(treenode *)CALLOC(1,sizeof(treenode),"procedure");
                  strcpy(templ->nodename, $2);
                  strcpy(templ->unitname, $3);
                add_sym(templ,symtab[TOS]);
              templ->rightchild = $4;  /* CONNECTS PROC W/PARMS */
                add_sym(templ,paramtab);
             }
      |      TOK_FUNCTION identifier unit_decl opt_formal_parm_list
opt_return
             {
                  TOS++;
                for (i = 0;i < MAX_TABSIZE; i++)
                 strcpy(symtab[TOS][i].nodename,EMPTY);
                templ=(treenode *)CALLOC(1,sizeof(treenode),"function");
                  strcpy(templ->nodename, $2);
                  strcpy(templ->unitname, $3);
                add_sym(templ,symtab[TOS]);
              templ->rightchild = $4;  /* CONNECTS FUNC W/PARMS */
              templ->leftchild = $5;
                add_sym(templ,paramtab);
             }
      ;
opt_formal_parm_list :
             TOK_OPENPAREN {temp2 = NULL;} formal_parms TOK_CLOSEPAREN
             { $$ = temp2; }
      |      { $$ = NULL; }

      ;
formal_parms :
             opt_var id_list TOK_COLON formal_parms_trailer
                 { current = $2;
                   templ = type_lookup($4);
                   while (current != NULL) {
                    add_sym(current,symtab[TOS]);
                if (templ != NULL)
                   build_sym(current->nodename, templ);
                    next = current;
                current = current->rightchild;
               }
                   if (next != NULL) {
                 next->rightchild = temp2;
                 temp2 = $2;
               }
               $$ = NULL; /* unused */
             }
      |      procfunction_heading proc_parm_trailer
                 { $$ = NULL; /* unused */}
      ;
opt_var :
             TOK_VAR
      |
      ;
formal_parms_trailer :
```

```
            identifier proc_parm_trailer {$$ = $1;}
      |     cas proc_parm_trailer {$$ = EMPTY;}
      ;
proc_parm_trailer :
            TOK_SEMICOLON formal_parms
      |
      ;
cas :
            opt_packed  TOK_OPENBRACKET  index_type_spec_list
TOK_CLOSEBRACKET TOK_OF identifier
      |     opt_packed  TOK_OPENBRACKET  index_type_spec_list
TOK_CLOSEBRACKET TOK_OF identifier cas
      ;
opt_packed :
            TOK_PACKED
      |
      ;
index_type_spec_list :
            identifier TOK_DOTDOT identifier TOK_COLON identifier
      ;
opt_return :
            TOK_COLON identifier
            {
              temp1 = type_lookup($2);
                  if (temp1 == NULL) {
                    temp1 = (treenode *)
                            CALLOC(1,sizeof(treenode),"opt_return");
                    strcpy(temp1->nodename,$2);
                }
              $$ = temp1;
            }
      | { $$ = NULL;}
      ;
type_dcls :
            identifier unit_decl TOK_EQUAL type TOK_SEMICOLON
            {
              temp1=(treenode *)CALLOC(1,sizeof(treenode),"type_dcls1");
              strcpy(temp1->nodename, $1);
              strcpy(temp1->unitname, $2);
                  temp1->leftchild = $4;
              add_sym(temp1,typetab);
            }
      |     type_dcls identifier unit_decl TOK_EQUAL type TOK_SEMICOLON
            {
              temp1=(treenode *)CALLOC(1,sizeof(treenode),"type_dcls2");
              strcpy(temp1->nodename, $2);
              strcpy(temp1->unitname, $3);
                  temp1->leftchild = $5;
              add_sym(temp1,typetab);
            }
      ;
identifier :
                { tmp=CALLOC((MAXSIZE),1,"identifier");
                  strncpy(tmp,yytext,MAXSIZE);
                }
```

```
          TOK_IDENTIFIER
          {$$ = tmp;}
     |
              { tmp=CALLOC((MAXSIZE),1,"identifier");
                strncpy(tmp,yytext,MAXSIZE);
          }
          standard_identifier
          {$$ = tmp;}
     ;
standard_identifier :
          TOK_BOOLEAN
     |    TOK_REAL
     |    TOK_INTEGER
     |    TOK_CHAR
     |    TOK_ABS
     |    TOK_ARCTAN
     |    TOK_ARGC
     |    TOK_ARGV
     |    TOK_CARD
     |    TOK_CHR
     |    TOK_CLOCK
     |    TOK_COS
     |    TOK_DATE
     |    TOK_DISPOSE
     |    TOK_EOF
     |    TOK_EOLN
     |    TOK_EXP
     |    TOK_EXPO
     |    TOK_FLUSH
     |    TOK_GET
     |    TOK_HALT
     |    TOK_LINELIMIT
     |    TOK_LN
     |    TOK_MESSAGE
     |    TOK_NEW
     |    TOK_NULL
     |    TOK_ODD
     |    TOK_ORD
     |    TOK_PACK
     |    TOK_PAGE
     |    TOK_PRED
     |    TOK_PUT
     |    TOK_RANDOM
     |    TOK_READ
     |    TOK_READLN
     |    TOK_REMOVE
     |    TOK_RESET
     |    TOK_REWRITE
     |    TOK_ROUND
     |    TOK_SEED
     |    TOK_SIN
     |    TOK_SQR
     |    TOK_SQRT
     |    TOK_STLIMIT
     |    TOK_SUCC
```

```
        |       TOK_SYSCLOCK
        |       TOK_TEXT
        |       TOK_TIME
        |       TOK_TRUNC
        |       TOK_UNDEFINED
        |       TOK_UNPACK
        |       TOK_WALLCLOCK
        |       TOK_WRITE
        |       TOK_WRITELN
        ;
%%
/*----------------     DECLARATIONS AND FUNCTIONS     ---------------*/
#include    <stdio.h>
#include    <ctype.h>
#include    <varargs.h>
#include    <string.h>
#define     TRUE  1
#define     FALSE 0
#define MAXSIZE 80
#define MAX_TABSIZE 857
#define EMPTY "\0"
#ifdef PRINTCALLOC
#define  CALLOC(Length,  Elts,  Location)    calloc(Length,
Elts);printf("calling calloc(%d) in %s\n", Length*Elts, Location)
#else
#define CALLOC(Length, Elts, Location) calloc(Length,Elts)
#endif
struct tnode
 {
  short marked;
  char  nodename[MAXSIZE];
  char  unitname[MAXSIZE];
  struct tnode *leftchild;
  struct tnode *rightchild;
 };
typedef struct tnode treenode;
extern  hashpjw();
extern      FILE  *yyin;
extern      char  yytext[];
extern      int   end_of_file;
extern      int   yylineno;
extern      char  *calloc();
extern  char    *strcat();
treenode typetab[MAX_TABSIZE];
treenode paramtab[MAX_TABSIZE];
treenode symtab[30][MAX_TABSIZE];
treenode *cur_parms[MAXSIZE];
treenode *deferred[300];
int      deftop;
int      parmtop;
int    TOS;
int    i;
int    temptype;
char   *tmp;
char     *tmp1;
```

```
char     *unit;
treenode *temp_id;
treenode *temp1;
treenode *temp2;
treenode *current;
treenode *temp_current;
treenode *next;
treenode *tempnode;
int    rhs;
#ifdef YYDEBUG
extern int yydebug;
#endif
/*--------------------      FUNCTIONS     --------------------*/
set_yydebug()
{
#      ifdef YYDEBUG
       yydebug = 1;
#      endif
}
mylex()
{
       int   token;
again:
       token = yylex();
#      ifdef YYDEBUG
       if (yydebug)
             printf("## %d [%d] |%s|\n",yylineno,token,yytext);
#      endif
       if((token == TOK_WHITESPACE) || (token == TOK_NEWLINE) || (token
== TOK_UNKNOWN))
             goto again;
         if ((token == TOK_COMMENT1_START)) {
           token = yylex();
#        ifdef YYDEBUG
         if (yydebug)
           printf("## %d [%d] |%s|\n",yylineno,token,yytext);
#        endif
           if (token == TOK_AMPERSAND) {
             return(TOK_UNIT1);
             }
           while (token != TOK_COMMENT1_END) {
           token = yylex();
#        ifdef YYDEBUG
         if (yydebug)
           printf("## %d [%d] |%s|\n",yylineno,token,yytext);
#        endif
           }
          goto again;
          }
         if ((token == TOK_COMMENT2_START)) {
           token = yylex();
#        ifdef YYDEBUG
         if (yydebug)
           printf("## %d [%d] |%s|\n",yylineno,token,yytext);
#        endif
```

```
          if (token == TOK_AMPERSAND) {
             return(TOK_UNIT2);
             }
          while (token != TOK_COMMENT2_END) {
          token = yylex();
#        ifdef YYDEBUG
         if (yydebug)
            printf("## %d [%d] |%s|\n",yylineno,token,yytext);
#        endif
         }
       goto again;
       }
#      ifdef TRACE_TOKENS
        printf("%3d:%s\n", token, yytext);
#      endif
       return(token);
}
int print_node(expr)
treenode *expr;
{
  int ok = TRUE;
  printf("%s ", expr->nodename);
  if (expr->leftchild == NULL && expr->rightchild == NULL)
    printf("%s\n", expr->unitname);
  else if(expr->leftchild != NULL && expr->rightchild == NULL)
    printf("1\n");
  else if(expr->rightchild != NULL && expr->leftchild == NULL)
    printf("1\n");
  else if (expr->leftchild != NULL && expr->rightchild != NULL)
    printf("2\n");
  if (ok && expr->leftchild != NULL)
  {
   ok = print_node(expr->leftchild);
  }
  if (ok && expr->rightchild != NULL)
  {
   ok = print_node(expr->rightchild);
  }
   if (!ok) return FALSE;
   else return TRUE;
}
void add_sym(entry, table)
treenode *entry;
treenode table[];
{
 int i, first_i,done;
 i = hashpjw(entry->nodename);
 first_i = i;
 done = 0;
 while (done == 0) {
  if (strcmp(table[i].nodename, EMPTY) != 0 ) i = (i + 1)%MAX_TABSIZE;
  else done = 1;
  if (i == first_i) done = 1;
  }
 if (strcmp(table[i].nodename, EMPTY) != 0)
```

```
      fprintf(stderr,"Warning, Symbol table exceeded\n");
 table[i] = *entry;
}
char *lookup(name)
char *name;
{
 int i,j,first_i,done;
 i = hashpjw(name);
 j = TOS;
 first_i = i;
 done = FALSE;
  while (done == FALSE)
  {
    if (strcmp(symtab[j][i].nodename, name) == 0)
      done = TRUE;
    else
      {
        i = (i + 1)%MAX_TABSIZE;
        if (i == first_i)
          j--;       /* right back where we started in stack */
        if (j < 0)   /* run off bottom of stack */
          done = TRUE;
      }
  }
  if (j < 0)           /* run off bottom of stack */
   return(EMPTY);
  else
    return(symtab[j][i].unitname);   /* return unit associated w/
identifier */
}
treenode *type_lookup(name)
char *name;
{
 int i, first_i,done;
 i = hashpjw(name);
 first_i = i;
 done = FALSE;
  while (done == FALSE)
  {
    if (strcmp(typetab[i].nodename, name) == 0)
      done = TRUE;
    else
      {
        i = (i + 1)%MAX_TABSIZE;
        if (i == first_i)
        done = TRUE;
      }
  }
  if (strcmp(typetab[i].nodename,name) != 0)
   return(NULL);
  else
   return(typetab[i].leftchild);
}
treenode *parm_lookup(proc_func_name)
char *proc_func_name;
```

```
{
 int i, first_i,done;
 i = hashpjw(proc_func_name);
 first_i = i;
 done = FALSE;
  while (done == FALSE)
  {
    if (strcmp(paramtab[i].nodename, proc_func_name) == 0)
      done = TRUE;
    else
      {
        i = (i + 1)%MAX_TABSIZE;
        if (i == first_i)
        done = TRUE;
      }
  }
  if (strcmp(paramtab[i].nodename,proc_func_name) != 0)
   return(NULL);
  else
   return(&(paramtab[i]));
}
int build_sym(str,head)
char *str;
 treenode *head;
{
 char *headstr;
 treenode *temp;
  if ( head == NULL ) {
#ifdef NULLLEAFPRINT
     printf("Null head on %s\n",str);
#endif
     return;
     }
  if (head->leftchild == NULL && head->rightchild == NULL) {
#ifdef NULLLEAFPRINT
     printf("leaf head on %s\n",str);
#endif
     return;
     }
  tempnode = (treenode *)CALLOC(1,sizeof(treenode),"build_sym");
  while (head != NULL)
  {
    strcpy(tempnode->nodename,str);
    strcpy(tempnode->unitname,head->unitname);
    strcat(tempnode->nodename,head->nodename);
    if (strcmp(head->nodename, ".") != 0)
      {
      tempnode->leftchild = NULL;
      tempnode->rightchild = NULL;
      add_sym(tempnode,symtab[TOS]);
      }
    build_sym(tempnode->nodename,head->leftchild);
    head = head->rightchild;
  }
}
```

```
void break_link(link) /* caused by linked-list declarations */
treenode *link;
{treenode *trav;
 if (link==NULL) return;
 trav=link;
 while (trav!=NULL) {
  trav->marked = 1;
  if (trav->leftchild != NULL)
    if (trav->leftchild->marked==1)
  {
    trav->leftchild = NULL;
  }
  else break_link(trav->leftchild);
  trav = trav->rightchild;
 }
}
void clear_mark(link)
treenode *link;
{treenode *trav;
 trav = link;
 while (trav != NULL) {
   if (trav->marked == 1) {
     trav->marked = 0;
     clear_mark(trav->leftchild);
     }
   trav = trav->rightchild;
   }
 }
#define      yylex mylex
```

```
/*---------------------------------------------------------------------*/
/*                                                                     */
/*                          pascalscan.l                               */
/*                                                                     */
/*              Scanner specification for Reacher                     */
/*                                                                     */
/*    Notes:                                                           */
/*    Null string is not allowed                                       */
/*    Comments longer than 954 characters explode the scanner         */
/*---------------------------------------------------------------------*/
%{
/* Declarations created by yacc for the tokens */
#include "y.tab.h"
%}
letter          [A-Za-z]
digit       [0-9]
underscore  "_"
litstring   ("'"([^\'\n]|"''")+"'")
/*---------------------------------------------------------------------*/
                            /*  RULES  */
%%
{letter}({letter}|{digit}|{underscore})*   { return(check_id(yytext)); }
{digit}+"."{digit}+[eE]"+"{digit}{digit}*  { return(TOK_UNSIGNED_REAL);
}
{digit}+"."{digit}+[eE]"-"{digit}{digit}*  { return(TOK_UNSIGNED_REAL);
}
{digit}+"."{digit}+[eE]{digit}{digit}*                              {
return(TOK_UNSIGNED_REAL); }
{digit}+"."{digit}+                     { return(TOK_UNSIGNED_REAL); }
{digit}+[eE]"+"{digit}+                { return(TOK_UNSIGNED_REAL); }
{digit}+[eE]"-"{digit}+                { return(TOK_UNSIGNED_REAL); }
{digit}+[eE]{digit}+                   { return(TOK_UNSIGNED_REAL); }
{digit}+                  { return(TOK_UNSIGNED_INTEGER); }
"{"                       { return(TOK_COMMENT1_START); }
"}"                       { return(TOK_COMMENT1_END); }
"(*"                      { return(TOK_COMMENT2_START); }
"*)"                      { return(TOK_COMMENT2_END); }
"&"                       { return(TOK_AMPERSAND);  }
"."                       { return(TOK_PERIOD); }
";"                       { return(TOK_SEMICOLON); }
"/"                       { return(TOK_DIVIDE); }
".."                      { return(TOK_DOTDOT); }
">"                       { return(TOK_GREATERTHAN); }
">="                      { return(TOK_GREATERTHANOREQUALTO); }
"<"                       { return(TOK_LESSTHAN); }
"<="                      { return(TOK_LESSTHANOREQUALTO); }
"<>"                      { return(TOK_NOTEQUAL); }
"-"                       { return(TOK_MINUS); }
"("                       { return(TOK_OPENPAREN); }
")"                       { return(TOK_CLOSEPAREN); }
"*"                       { return(TOK_MULT); }
"["                       { return(TOK_OPENBRACKET); }
"]"                       { return(TOK_CLOSEBRACKET); }
"+"                       { return(TOK_PLUS); }
[@\^]                     { return(TOK_POINTER); }
```

```
":"                     { return(TOK_COLON); }
","                     { return(TOK_COMMA); }
"="                     { return(TOK_EQUAL); }
":="                    { return(TOK_ASSIGN); }
{litstring}             { return(TOK_STRING); }
[ \t\f]+                { return(TOK_WHITESPACE); }
[\n]                    { return(TOK_NEWLINE); }
.                       { return(TOK_UNKNOWN); }
%%
/*----------------------------------------------------------------------*/
                        /*  USER SUBROUTINES  */
#include <stdio.h>
#include <ctype.h>
#include <varargs.h>
#define     TRUE  1
#define     FALSE 0
/***********************************************************************
Data structures for pre-defined identifiers:
      struct rsvd[]                 Reserved words
      struct std_type[]       Standard or predefined types
      struct std_func[]       Standard functions
      struct std_proc[]       Standard procedures
***********************************************************************/
/***********     RESERVED WORD Structure     ***********/
struct      {
                  char  *s;
                  int         code;
            } rsvd[] = {
            { "AND",          TOK_AND },
            { "ARRAY",        TOK_ARRAY },
            { "BEGIN",        TOK_BEGIN },
            { "CASE",         TOK_CASE },
            { "CONST",        TOK_CONST },
            { "DIV",          TOK_DIV },
            { "DO",                 TOK_DO },
            { "DOWNTO",       TOK_DOWNTO },
            { "ELSE",         TOK_ELSE },
            { "END",          TOK_END },
            { "FILE",         TOK_FILE },
            { "FOR",          TOK_FOR },
            { "FORWARD",      TOK_FORWARD },
            { "FUNCTION",     TOK_FUNCTION },
            { "GOTO",         TOK_GOTO },
            { "IF",                 TOK_IF },
            { "IN",                 TOK_IN },
            { "LABEL",        TOK_LABEL },
            { "MOD",          TOK_MOD },
            { "NIL",          TOK_NIL },
            { "NOT",          TOK_NOT },
            { "OF",                 TOK_OF },
            { "OR",                 TOK_OR },
            { "PACKED",       TOK_PACKED },
            { "PROCEDURE",    TOK_PROCEDURE },
            { "PROGRAM",      TOK_PROGRAM },
            { "RECORD",       TOK_RECORD },
```

```
		{ "REPEAT",		TOK_REPEAT },
		{ "SET",		TOK_SET },
		{ "THEN",		TOK_THEN },
		{ "TO",				TOK_TO },
		{ "TYPE",		TOK_TYPE },
		{ "UNTIL",		TOK_UNTIL },
		{ "VAR",		TOK_VAR },
		{ "WHILE",		TOK_WHILE },
		{ "WITH",		TOK_WITH },
		{ NULL,				0 }
		};
struct		{
			char	*s;
			int		code;
		} std_type[] = {
		{ "BOOLEAN",		TOK_BOOLEAN },
		{ "INTEGER",		TOK_INTEGER },
		{ "REAL",		TOK_REAL },
/*		{ "STRING", TOK_STRING },*/	/* ISO standard?? */
		{ "TEXT",		TOK_TEXT },
		{ NULL,				0 }
		};
struct		{
			char	*s;
			int		code;
		}	std_func[] = {
		{ "ABS",		TOK_ABS },
		{ "SQR",		TOK_SQR },
		{ "SQRT",		TOK_SQRT },
		{ "SIN",		TOK_SIN },
		{ "COS",		TOK_COS },
		{ "ARCTAN",		TOK_ARCTAN },
		{ "LN",				TOK_LN },
		{ "EXP",		TOK_EXP },
		{ "TRUNC",		TOK_TRUNC },
		{ "ROUND",		TOK_ROUND },
		{ "ORD",		TOK_ORD },
		{ "CHR",		TOK_CHR },
		{ "SUCC",		TOK_SUCC },
		{ "PRED",		TOK_PRED },
		{ "ODD",		TOK_ODD },
		{ "EOLN",		TOK_EOLN },
		{ "EOF",		TOK_EOF },
		{ "ARGC",		TOK_ARGC }, /* Berkeley Pascal Standard
Functions */
		{ "CARD",		TOK_CARD },	/* | */
		{ "CLOCK",		TOK_CLOCK },	/* | */
		{ "EXPO",		TOK_EXPO },	/*\ /*/
		{ "RANDOM",		TOK_RANDOM },	/* . */
		{ "SEED",		TOK_SEED },
		{ "SYSCLOCK",		TOK_SYSCLOCK },
		{ "UNDEFINED",		TOK_UNDEFINED },
		{ "WALLCLOCK",		TOK_WALLCLOCK },
		{ NULL,				0 }
		};
```

```
struct      {
                  char  *s;
                  int         code;
            }     std_proc[] = {
            { "READ",         TOK_READ },
            { "READLN",       TOK_READLN },
            { "WRITE",        TOK_WRITE },
            { "WRITELN",      TOK_WRITELN },
            { "REWRITE",      TOK_REWRITE },
            { "RESET",        TOK_RESET },
            { "PUT",          TOK_PUT },
            { "GET",          TOK_GET },
            { "PAGE",         TOK_PAGE },
            { "NEW",          TOK_NEW },
            { "DISPOSE",      TOK_DISPOSE },
            { "PACK",         TOK_PACK },
            { "UNPACK",       TOK_UNPACK },
            { "ARGV",         TOK_ARGV },/*Bkly Pascal Std Procedures */
            { "DATE",         TOK_DATE },       /* | */
            { "FLUSH",        TOK_FLUSH },      /* | */
            { "HALT",         TOK_HALT },       /*\ /*/
            { "LINELIMIT",    TOK_LINELIMIT },  /* . */
            { "MESSAGE",      TOK_MESSAGE },
            { "NULL",         TOK_NULL },
            { "REMOVE",       TOK_REMOVE },
            { "STLIMIT",      TOK_STLIMIT },
            { "TIME",         TOK_TIME },
            { NULL,                 0 }
            };
check_id(t)
char  *t;
{
      register i;
      char  s[256];
      strcpy(s,t);
      for (i=0; s[i]; i++) {
            if (islower(s[i]))
                  s[i] = toupper(s[i]);
      }
      /* First see if this is a reserved word */
      for (i=0; rsvd[i].s; i++) {
            if (!strcmp(s,rsvd[i].s))
                  return(rsvd[i].code);
      }
      /* Now see if this is a standard type */
      for (i=0; std_type[i].s; i++) {
            if (!strcmp(s, std_type[i].s))
                  return(std_type[i].code);
      }
      /* Now see if this is a standard function */
      for (i=0; std_func[i].s; i++) {
            if (!strcmp(s, std_func[i].s))
                  return(std_func[i].code);
      }
      /* Now see if this is a standard procedure */
```

```
    for (i=0; std_proc[i].s; i++) {
        if (!strcmp(s, std_proc[i].s))
            return(std_proc[i].code);
    }
    /* Default - plain old identifier */
    return(TOK_IDENTIFIER);
}
```

/* Pparse.c */

```
# include "stdio.h"
# define U(x) x
# define NLSTATE yyprevious=YYNEWLINE
# define BEGIN yybgin = yysvec + 1 +
# define INITIAL 0
# define YYLERR yysvec
# define YYSTATE (yyestate-yysvec-1)
# define YYOPTIM 1
# define YYLMAX BUFSIZ
# define output(c) putc(c,yyout)
#   define   input()   (((yytchar=yysptr>yysbuf?U(*--
yysptr):getc(yyin))==10?(yylineno++,yytchar):yytchar)==EOF?0:yytchar)
#  define  unput(c)  {yytchar=  (c);if(yytchar=='\n')yylineno--
;*yysptr++=yytchar;}
# define yymore() (yymorfg=1)
# define ECHO fprintf(yyout, "%s",yytext)
# define REJECT { nstr = yyreject(); goto yyfussy;}
int yyleng; extern char yytext[];
int yymorfg;
extern char *yysptr, yysbuf[];
int yytchar;
FILE *yyin = {stdin}, *yyout = {stdout};
extern int yylineno;
struct yysvf {
      struct yywork *yystoff;
      struct yysvf *yyother;
      int *yystops;};
struct yysvf *yyestate;
extern struct yysvf yysvec[], *yybgin;
/* Declarations created by yacc for the tokens */
#include "y.tab.h"
                                  /*  RULES  */
# define YYNEWLINE 10
yylex(){
int nstr; extern int yyprevious;
while((nstr = yylook()) >= 0)
yyfussy: switch(nstr){
case 0:
if(yywrap()) return(0); break;
case 1:
  { return(check_id(yytext)); }
break;
case 2:
 { return(TOK_UNSIGNED_REAL); }
break;
case 3:
 { return(TOK_UNSIGNED_REAL); }
break;
case 4:
   { return(TOK_UNSIGNED_REAL); }
break;
case 5:
```

```
                { return(TOK_UNSIGNED_REAL); }
break;
case 6:
               { return(TOK_UNSIGNED_REAL); }
break;
case 7:
               { return(TOK_UNSIGNED_REAL); }
break;
case 8:
               { return(TOK_UNSIGNED_REAL); }
break;
case 9:
             { return(TOK_UNSIGNED_INTEGER); }
break;
case 10:
                   { return(TOK_COMMENT1_START); }
break;
case 11:
                   { return(TOK_COMMENT1_END); }
break;
case 12:
                   { return(TOK_COMMENT2_START); }
break;
case 13:
                   { return(TOK_COMMENT2_END); }
break;
case 14:
                   { return(TOK_AMPERSAND);  }
break;
case 15:
                   { return(TOK_PERIOD); }
break;
case 16:
                   { return(TOK_SEMICOLON); }
break;
case 17:
                   { return(TOK_DIVIDE); }
break;
case 18:
                   { return(TOK_DOTDOT); }
break;
case 19:
                   { return(TOK_GREATERTHAN); }
break;
case 20:
                   { return(TOK_GREATERTHANOREQUALTO); }
break;
case 21:
                   { return(TOK_LESSTHAN); }
break;
case 22:
                   { return(TOK_LESSTHANOREQUALTO); }
break;
case 23:
                   { return(TOK_NOTEQUAL); }
```

```
break;
case 24:
                        { return(TOK_MINUS); }
break;
case 25:
                        { return(TOK_OPENPAREN); }
break;
case 26:
                        { return(TOK_CLOSEPAREN); }
break;
case 27:
                        { return(TOK_MULT); }
break;
case 28:
                        { return(TOK_OPENBRACKET); }
break;
case 29:
                        { return(TOK_CLOSEBRACKET); }
break;
case 30:
                        { return(TOK_PLUS); }
break;
case 31:
                        { return(TOK_POINTER); }
break;
case 32:
                        { return(TOK_COLON); }
break;
case 33:
                        { return(TOK_COMMA); }
break;
case 34:
                        { return(TOK_EQUAL); }
break;
case 35:
                        { return(TOK_ASSIGN); }
break;
case 36:
                { return(TOK_STRING); }
break;
case 37:
                { return(TOK_WHITESPACE); }
break;
case 38:
                        { return(TOK_NEWLINE); }
break;
case 39:
                        { return(TOK_UNKNOWN); }
break;
case -1:
break;
default:
fprintf(yyout,"bad switch yylook %d",nstr);
} return(0); }
/* end of yylex */
```

```
/*---------------------------------------------------------------------
*/
                                /*  USER SUBROUTINES  */
#include <stdio.h>
#include <ctype.h>
#include <varargs.h>
#define     TRUE  1
#define     FALSE 0
/*********************************************************************
Data structures for pre-defined identifiers:
      struct rsvd[]                 Reserved words
      struct std_type[]       Standard or predefined types
      struct std_func[]       Standard functions
      struct std_proc[]       Standard procedures
*********************************************************************/
/***********     RESERVED WORD Structure     ***********/
struct      {
                  char  *s;
                  int         code;
            } rsvd[] = {
            { "AND",          TOK_AND },
            { "ARRAY",        TOK_ARRAY },
            { "BEGIN",        TOK_BEGIN },
            { "CASE",         TOK_CASE },
            { "CONST",        TOK_CONST },
            { "DIV",          TOK_DIV },
            { "DO",                 TOK_DO },
            { "DOWNTO",       TOK_DOWNTO },
            { "ELSE",         TOK_ELSE },
            { "END",          TOK_END },
            { "FILE",         TOK_FILE },
            { "FOR",          TOK_FOR },
            { "FORWARD",      TOK_FORWARD },
            { "FUNCTION",     TOK_FUNCTION },
            { "GOTO",         TOK_GOTO },
            { "IF",                 TOK_IF },
            { "IN",                 TOK_IN },
            { "LABEL",        TOK_LABEL },
            { "MOD",          TOK_MOD },
            { "NIL",          TOK_NIL },
            { "NOT",          TOK_NOT },
            { "OF",                 TOK_OF },
            { "OR",                 TOK_OR },
            { "PACKED",       TOK_PACKED },
            { "PROCEDURE",    TOK_PROCEDURE },
            { "PROGRAM",      TOK_PROGRAM },
            { "RECORD",       TOK_RECORD },
            { "REPEAT",       TOK_REPEAT },
            { "SET",          TOK_SET },
            { "THEN",         TOK_THEN },
            { "TO",                 TOK_TO },
            { "TYPE",         TOK_TYPE },
            { "UNTIL",        TOK_UNTIL },
            { "VAR",          TOK_VAR },
            { "WHILE",        TOK_WHILE },
```

```
        { "WITH",          TOK_WITH },
        { NULL,                 0 }
        };
struct  {
                char  *s;
                int         code;
        } std_type[] = {
        { "BOOLEAN",       TOK_BOOLEAN },
        { "INTEGER",       TOK_INTEGER },
        { "REAL",        TOK_REAL },
/*      { "STRING", TOK_STRING },*/   /* ISO standard?? */
        { "TEXT",          TOK_TEXT },
        { NULL,                 0 }
        };
struct  {
                char  *s;
                int         code;
        }       std_func[] = {
        { "ABS",           TOK_ABS },
        { "SQR",           TOK_SQR },
        { "SQRT",          TOK_SQRT },
        { "SIN",           TOK_SIN },
        { "COS",           TOK_COS },
        { "ARCTAN",        TOK_ARCTAN },
        { "LN",                 TOK_LN },
        { "EXP",           TOK_EXP },
        { "TRUNC",         TOK_TRUNC },
        { "ROUND",         TOK_ROUND },
        { "ORD",           TOK_ORD },
        { "CHR",           TOK_CHR },
        { "SUCC",          TOK_SUCC },
        { "PRED",          TOK_PRED },
        { "ODD",           TOK_ODD },
        { "EOLN",          TOK_EOLN },
        { "EOF",           TOK_EOF },
        { "ARGC",          TOK_ARGC }, /* Berkeley Pascal Standard
Functions */
        { "CARD",          TOK_CARD },     /* | */
        { "CLOCK",         TOK_CLOCK },    /* | */
        { "EXPO",          TOK_EXPO },     /*\ /*/
        { "RANDOM",        TOK_RANDOM },   /* . */
        { "SEED",          TOK_SEED },
        { "SYSCLOCK",      TOK_SYSCLOCK },
        { "UNDEFINED",     TOK_UNDEFINED },
        { "WALLCLOCK",     TOK_WALLCLOCK },
        { NULL,                 0 }
        };
struct  {
                char  *s;
                int         code;
        }       std_proc[] = {
        { "READ",          TOK_READ },
        { "READLN",        TOK_READLN },
        { "WRITE",         TOK_WRITE },
        { "WRITELN",       TOK_WRITELN },
```

```
        { "REWRITE",      TOK_REWRITE },
        { "RESET",        TOK_RESET },
        { "PUT",          TOK_PUT },
        { "GET",          TOK_GET },
        { "PAGE",         TOK_PAGE },
        { "NEW",          TOK_NEW },
        { "DISPOSE",      TOK_DISPOSE },
        { "PACK",         TOK_PACK },
        { "UNPACK",       TOK_UNPACK },
        { "ARGV",         TOK_ARGV }, /* Berkeley Pascal Standard
Procedures */
        { "DATE",         TOK_DATE }, /* | */
        { "FLUSH",        TOK_FLUSH },      /* | */
        { "HALT",         TOK_HALT }, /*\ /*/
        { "LINELIMIT",    TOK_LINELIMIT },  /* . */
        { "MESSAGE",      TOK_MESSAGE },
        { "NULL",         TOK_NULL },
        { "REMOVE",       TOK_REMOVE },
        { "STLIMIT",      TOK_STLIMIT },
        { "TIME",         TOK_TIME },
        { NULL,                0 }
        };
check_id(t)
char  *t;
{
    register i;
    char  s[256];
    strcpy(s,t);
    for (i=0; s[i]; i++) {
        if (islower(s[i]))
            s[i] = toupper(s[i]);
    }
    /* First see if this is a reserved word */
    for (i=0; rsvd[i].s; i++) {
        if (!strcmp(s,rsvd[i].s))
            return(rsvd[i].code);
    }
    /* Now see if this is a standard type */
    for (i=0; std_type[i].s; i++) {
        if (!strcmp(s, std_type[i].s))
            return(std_type[i].code);
    }
    /* Now see if this is a standard function */
    for (i=0; std_func[i].s; i++) {
        if (!strcmp(s, std_func[i].s))
            return(std_func[i].code);
    }
    /* Now see if this is a standard procedure */
    for (i=0; std_proc[i].s; i++) {
        if (!strcmp(s, std_proc[i].s))
            return(std_proc[i].code);
    }
    /* Default - plain old identifier */
    return(TOK_IDENTIFIER);
}
```

```
int yyvstop[] = {
0,
39,
0,
37,
39,
0,
38,
0,
14,
39,
0,
39,
0,
25,
39,
0,
26,
39,
0,
27,
39,
0,
30,
39,
0,
33,
39,
0,
24,
39,
0,
15,
39,
0,
17,
39,
0,
9,
39,
0,
32,
39,
0,
16,
39,
0,
21,
39,
0,
34,
39,
0,
19,
39,
```

```
0,
31,
39,
0,
1,
39,
0,
28,
39,
0,
29,
39,
0,
10,
39,
0,
11,
39,
0,
37,
0,
12,
0,
13,
0,
18,
0,
9,
0,
35,
0,
22,
0,
23,
0,
20,
0,
1,
0,
36,
0,
5,
0,
8,
0,
6,
0,
7,
0,
4,
0,
2,
0,
3,
0,
```

```
0};
# define YYTYPE char
struct yywork { YYTYPE verify, advance; } yycrank[] = {
0,0,  0,0,  1,3,  0,0,
0,0,  0,0,  0,0,  0,0,
0,0,  0,0,  1,4,  1,5,
4,28, 0,0,  0,0,  4,28,
0,0,  0,0,  0,0,  0,0,
0,0,  0,0,  0,0,  0,0,
29,0, 0,0,  0,0,  0,0,
0,0,  0,0,  0,0,  0,0,
0,0,  0,0,  0,0,  4,28,
0,0,  0,0,  0,0,  1,6,
1,7,  1,8,  1,9,  1,10,
1,11, 1,12, 1,13, 1,14,
1,15, 1,16, 8,31, 10,32,
14,33,      29,42,      30,29,      0,0,
0,0,  0,0,  0,0,  1,17,
1,18, 1,19, 1,20, 1,21,
17,37,      1,22, 1,23, 19,38,
19,39,      21,40,      1,23, 43,47,
0,0,  0,0,  0,0,  2,6,
0,0,  2,8,  2,9,  2,10,
2,11, 2,12, 2,13, 2,14,
2,15, 0,0,  0,0,  0,0,
0,0,  0,0,  0,0,  0,0,
1,24, 7,29, 1,25, 2,17,
2,18, 2,19, 2,20, 2,21,
0,0,  7,29, 7,0,  43,47,
16,34,      0,0,  16,35,      16,35,
16,35,      16,35,      16,35,      16,35,
16,35,      16,35,      16,35,      16,35,
0,0,  0,0,  0,0,  0,0,
0,0,  0,0,  0,0,  0,0,
1,26, 0,0,  1,27, 16,36,
2,24, 0,0,  2,25, 7,30,
0,0,  0,0,  0,0,  0,0,
0,0,  0,0,  0,0,  0,0,
7,29, 0,0,  0,0,  0,0,
0,0,  0,0,  23,41,      23,41,
23,41,      23,41,      23,41,      23,41,
23,41,      23,41,      23,41,      23,41,
7,29, 7,29, 0,0,  16,36,
2,26, 7,29, 2,27, 23,41,
23,41,      23,41,      23,41,      23,41,
23,41,      23,41,      23,41,      23,41,
23,41,      23,41,      23,41,      23,41,
23,41,      23,41,      23,41,      23,41,
23,41,      23,41,      23,41,      23,41,
23,41,      23,41,      23,41,      23,41,
23,41,      0,0,  0,0,  0,0,
0,0,  23,41,      0,0,  23,41,
23,41,      23,41,      23,41,      23,41,
23,41,      23,41,      23,41,      23,41,
23,41,      23,41,      23,41,      23,41,
```

```
23,41,      23,41,      23,41,      23,41,
23,41,      23,41,      23,41,      23,41,
23,41,      23,41,      23,41,      23,41,
23,41,      34,43,      34,43,      34,43,
34,43,      34,43,      34,43,      34,43,
34,43,      34,43,      34,43,      36,44,
0,0,  36,45,      0,0,  0,0,
36,46,      36,46,      36,46,      36,46,
36,46,      36,46,      36,46,      36,46,
36,46,      36,46,      44,48,      44,48,
44,48,      44,48,      44,48,      44,48,
44,48,      44,48,      44,48,      44,48,
45,49,      45,49,      45,49,      45,49,
45,49,      45,49,      45,49,      45,49,
45,49,      45,49,      46,46,      46,46,
46,46,      46,46,      46,46,      46,46,
46,46,      46,46,      46,46,      46,46,
47,50,      0,0,  47,51,      0,0,
0,0,  47,52,      47,52,      47,52,
47,52,      47,52,      47,52,      47,52,
47,52,      47,52,      47,52,      50,53,
50,53,      50,53,      50,53,      50,53,
50,53,      50,53,      50,53,      50,53,
50,53,      51,54,      51,54,      51,54,
51,54,      51,54,      51,54,      51,54,
51,54,      51,54,      51,54,      52,52,
52,52,      52,52,      52,52,      52,52,
52,52,      52,52,      52,52,      52,52,
52,52,      0,0,  0,0,  0,0,
0,0};
struct yysvf yysvec[] = {
0,    0,    0,
yycrank+-1, 0,          0,
yycrank+-37,      yysvec+1,   0,
yycrank+0,  0,          yyvstop+1,
yycrank+3,  0,          yyvstop+3,
yycrank+0,  0,          yyvstop+6,
yycrank+0,  0,          yyvstop+8,
yycrank+-92,      0,          yyvstop+11,
yycrank+8,  0,          yyvstop+13,
yycrank+0,  0,          yyvstop+16,
yycrank+10, 0,          yyvstop+19,
yycrank+0,  0,          yyvstop+22,
yycrank+0,  0,          yyvstop+25,
yycrank+0,  0,          yyvstop+28,
yycrank+6,  0,          yyvstop+31,
yycrank+0,  0,          yyvstop+34,
yycrank+58, 0,          yyvstop+37,
yycrank+3,  0,          yyvstop+40,
yycrank+0,  0,          yyvstop+43,
yycrank+6,  0,          yyvstop+46,
yycrank+0,  0,          yyvstop+49,
yycrank+8,  0,          yyvstop+52,
yycrank+0,  0,          yyvstop+55,
yycrank+98, 0,          yyvstop+58,
```

```
yycrank+0,  0,          yyvstop+61,
yycrank+0,  0,          yyvstop+64,
yycrank+0,  0,          yyvstop+67,
yycrank+0,  0,          yyvstop+70,
yycrank+0,  yysvec+4,   yyvstop+73,
yycrank+-14,      yysvec+7,   0,
yycrank+15, 0,          0,
yycrank+0,  0,          yyvstop+75,
yycrank+0,  0,          yyvstop+77,
yycrank+0,  0,          yyvstop+79,
yycrank+173,      0,          0,
yycrank+0,  yysvec+16,  yyvstop+81,
yycrank+188,      0,          0,
yycrank+0,  0,          yyvstop+83,
yycrank+0,  0,          yyvstop+85,
yycrank+0,  0,          yyvstop+87,
yycrank+0,  0,          yyvstop+89,
yycrank+0,  yysvec+23,  yyvstop+91,
yycrank+0,  yysvec+30,  yyvstop+93,
yycrank+2,  yysvec+34,  yyvstop+95,
yycrank+198,      0,          0,
yycrank+208,      0,          0,
yycrank+218,      0,          yyvstop+97,
yycrank+233,      0,          0,
yycrank+0,  yysvec+44,  yyvstop+99,
yycrank+0,  yysvec+45,  yyvstop+101,
yycrank+243,      0,          0,
yycrank+253,      0,          0,
yycrank+263,      0,          yyvstop+103,
yycrank+0,  yysvec+50,  yyvstop+105,
yycrank+0,  yysvec+51,  yyvstop+107,
0,    0,    0};
struct yywork *yytop = yycrank+320;
struct yysvf *yybgin = yysvec+1;
char yymatch[] = {
00  ,01  ,01  ,01  ,01  ,01  ,01  ,01  ,
01  ,011 ,012 ,01  ,011 ,01  ,01  ,01  ,
01  ,01  ,01  ,01  ,01  ,01  ,01  ,01  ,
01  ,01  ,01  ,01  ,01  ,01  ,01  ,01  ,
011 ,01  ,01  ,01  ,01  ,01  ,01  ,047 ,
01  ,01  ,01  ,01  ,01  ,01  ,01  ,01  ,
'0' ,'0' ,'0' ,'0' ,'0' ,'0' ,'0' ,'0' ,
'0' ,'0' ,01  ,01  ,01  ,01  ,01  ,01  ,
'@' ,'A' ,'A' ,'A' ,'A' ,'E' ,'A' ,'A' ,
'A' ,'A' ,'A' ,'A' ,'A' ,'A' ,'A' ,'A' ,
'A' ,'A' ,'A' ,'A' ,'A' ,'A' ,'A' ,'A' ,
'A' ,'A' ,'A' ,01  ,01  ,01  ,'@' ,01  ,
01  ,'A' ,'A' ,'A' ,'A' ,'E' ,'A' ,'A' ,
'A' ,'A' ,'A' ,'A' ,'A' ,'A' ,'A' ,'A' ,
'A' ,'A' ,'A' ,'A' ,'A' ,'A' ,'A' ,'A' ,
'A' ,'A' ,'A' ,01  ,01  ,01  ,01  ,01  ,
0};
char yyextra[] = {
0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,
```

```
0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,
0};
#ifndef lint
static      char ncform_sccsid[] = "@(#)ncform 1.6 88/02/08 SMI"; /*
from S5R2 1.2 */
#endif
int yylineno =1;
# define YYU(x) x
# define NLSTATE yyprevious=YYNEWLINE
char yytext[YYLMAX];
struct yysvf *yylstate [YYLMAX], **yylsp, **yyolsp;
char yysbuf[YYLMAX];
char *yysptr = yysbuf;
int *yyfnd;
extern struct yysvf *yyestate;
int yyprevious = YYNEWLINE;
yylook(){
      register struct yysvf *yystate, **lsp;
      register struct yywork *yyt;
      struct yysvf *yyz;
      int yych, yyfirst;
      struct yywork *yyr;
# ifdef LEXDEBUG
      int debug;
# endif
      char *yylastch;
      /* start off machines */
# ifdef LEXDEBUG
      debug = 0;
# endif
      yyfirst=1;
      if (!yymorfg)
            yylastch = yytext;
      else {
            yymorfg=0;
            yylastch = yytext+yyleng;
            }
      for(;;){
            lsp = yylstate;
            yyestate = yystate = yybgin;
            if (yyprevious==YYNEWLINE) yystate++;
            for (;;){
# ifdef LEXDEBUG
                  if(debug)fprintf(yyout,"state %d\n",yystate-yysvec-1);
# endif
                  yyt = yystate->yystoff;
                  if(yyt == yycrank && !yyfirst){  /* may not be any
transitions */
                        yyz = yystate->yyother;
                        if(yyz == 0)break;
                        if(yyz->yystoff == yycrank)break;
                        }
                  *yylastch++ = yych = input();
```

```
                  yyfirst=0;
            tryagain:
# ifdef LEXDEBUG
                  if(debug){
                        fprintf(yyout,"char ");
                        allprint(yych);
                        putchar('\n');
                        }
# endif
                  yyr = yyt;
                  if ( (int)yyt > (int)yycrank){
                        yyt = yyr + yych;
                        if (yyt <= yytop && yyt->verify+yysvec ==
yystate){
                              if(yyt->advance+yysvec == YYLERR)   /*
error transitions */
                                    {unput(*--yylastch);break;}
                              *lsp++ = yystate = yyt->advance+yysvec;
                              goto contin;
                              }
                        }
# ifdef YYOPTIM
                  else if((int)yyt < (int)yycrank) {        /*   r   <
yycrank */
                        yyt = yyr = yycrank+(yycrank-yyt);
# ifdef LEXDEBUG
                        if(debug)fprintf(yyout,"compressed state\n");
# endif
                        yyt = yyt + yych;
                        if(yyt <= yytop && yyt->verify+yysvec ==
yystate){
                              if(yyt->advance+yysvec == YYLERR)   /*
error transitions */
                                    {unput(*--yylastch);break;}
                              *lsp++ = yystate = yyt->advance+yysvec;
                              goto contin;
                              }
                        yyt = yyr + YYU(yymatch[yych]);
# ifdef LEXDEBUG
                        if(debug){
                              fprintf(yyout,"try fall back character ");
                              allprint(YYU(yymatch[yych]));
                              putchar('\n');
                              }
# endif
                        if(yyt <= yytop && yyt->verify+yysvec ==
yystate){
                              if(yyt->advance+yysvec == YYLERR)   /*
error transition */
                                    {unput(*--yylastch);break;}
                              *lsp++ = yystate = yyt->advance+yysvec;
                              goto contin;
                              }
                        }
```

```
                    if ((yystate = yystate->yyother) && (yyt= yystate-
>yystoff) != yycrank){
# ifdef LEXDEBUG
                              if(debug)fprintf(yyout,"fall back to state
%d\n",yystate-yysvec-1);
# endif
                              goto tryagain;
                              }
# endif
                    else
                              {unput(*--yylastch);break;}
          contin:
# ifdef LEXDEBUG
                    if(debug){
                              fprintf(yyout,"state %d char ",yystate-yysvec-
1);
                              allprint(yych);
                              putchar('\n');
                              }
# endif
                    ;
                    }
# ifdef LEXDEBUG
          if(debug){
                    fprintf(yyout,"stopped at %d with ",*(lsp-1)-yysvec-
1);
                    allprint(yych);
                    putchar('\n');
                    }
# endif
          while (lsp-- > yylstate){
                    *yylastch-- = 0;
                    if (*lsp != 0 && (yyfnd= (*lsp)->yystops) && *yyfnd >
0){
                              yyolsp = lsp;
                              if(yyextra[*yyfnd]){          /* must backup */
                                        while(yyback((*lsp)->yystops,-*yyfnd) != 1
&& lsp > yylstate){
                                                  lsp--;
                                                  unput(*yylastch--);
                                                  }
                                        }
                              yyprevious = YYU(*yylastch);
                              yylsp = lsp;
                              yyleng = yylastch-yytext+1;
                              yytext[yyleng] = 0;
# ifdef LEXDEBUG
                              if(debug){
                                        fprintf(yyout,"\nmatch ");
                                        sprint(yytext);
                                        fprintf(yyout," action %d\n",*yyfnd);
                                        }
# endif
                              return(*yyfnd++);
                              }
```

```
                    unput(*yylastch);
                    }
              if (yytext[0] == 0  /* && feof(yyin) */)
                    {
                    yysptr=yysbuf;
                    return(0);
                    }
              yyprevious = yytext[0] = input();
              if (yyprevious>0)
                    output(yyprevious);
              yylastch=yytext;
# ifdef LEXDEBUG
              if(debug)putchar('\n');
# endif
              }
        }
yyback(p, m)
        int *p;
{
if (p==0) return(0);
while (*p)
        {
        if (*p++ == m)
              return(1);
        }
return(0);
}
        /* the following are only used in the lex library */
yyinput(){
        return(input());
        }
yyoutput(c)
  int c; {
        output(c);
        }
yyunput(c)
   int c; {
        unput(c);
        }
```

APPENDIX B

UNITCHECK PROCEDURE

```
#include <stdio.h>
#define END_OF_FP1  0
#define BEGIN_EXPN  1
#define NORMAL      2
#define INSERT_NODE 3
#define END_EXPN    4
#define FUNNY       5
#define MAXSIZE    80
#define MAXLINE   255
#define MAXFILE    80
#define TRUE        1
#define FALSE       0
#define EMPTY "\0"
#include <string.h>

struct tnode
 {
  char  nodename[MAXSIZE];
  char  unitname[MAXSIZE];
  struct tnode *leftchild;
  struct tnode *rightchild;
 };

typedef struct tnode treenode;
struct treelist
 {
  treenode *tree;
  struct treelist *next;
 };
typedef struct treelist tlist;
int read_exp(/*FILE *fp, treenode *expn  */);
tlist *load_list(/* FILE *fp2 */);
int line_no;
char start_line[255];
main(argc,argv)
int argc;
char *argv[];
{
 char in_file[MAXFILE],
      out_file[MAXFILE];
 int  read_status;
 treenode *exp;
 tlist *root;
 FILE *fp1,*fp2;
 if (argc < 3)
 {
```

```
  printf("usage:  %s <input_filename> <output filename>\n", argv[0]);
  exit(1);
 }
 strcpy(in_file,argv[1]);
 strcpy(out_file,argv[2]);
 fp1 = fopen(in_file,"r");
 fp2 = fopen(out_file,"r");
 if ((!fp1) || (!fp2))
 {
  printf("%s:  couldn't open files\n", argv[0]);
  exit(1);
 }
 line_no = 0;
 root = load_list(fp2);
 line_no = 0;
#ifdef PRINTREAD
  printf("Below is the exp being compared to each exp in linklist\n\n");
#endif
 while (!feof(fp1))
 {
  read_status = read_exp(fp1, &exp);
  switch(read_status) {
    case END_OF_FP1   :
    case BEGIN_EXPN   :
    case NORMAL       :
    case INSERT_NODE  :
      break;
    case END_EXPN     :
      match(root, exp);
      break;
    case FUNNY        :
      printf("main():  Something is funny.\n");
      break;
  } /* end switch() */
 } /* end while() */
 fclose(fp1);
 fclose(fp2);
}

int read_exp(fp1, expn)
FILE *fp1;
treenode **expn;
{
 char line[MAXLINE];
 char operator[MAXSIZE];
 char operand[MAXSIZE];
 char unit[MAXSIZE];
 int  temp,
      numb_arguments;
 treenode  *op_struct;
 treenode  *opnd_struct;

 if (fgets(line, MAXLINE, fp1) == NULL)
 {
  return(END_OF_FP1);
```

```
 }
 else {
 line_no++;
 if (line[0] == '{')
 {
#ifdef PRINTREAD
  printf(" Begin expression\n{\n");
#endif
  strcpy(start_line,line);
  return(BEGIN_EXPN);
 }
 else if (line[0] == '}')
 {
#ifdef PRINTREAD
  printf("}\n End expression\n");
#endif
  return(END_EXPN);
 }
 else if (sscanf(line,"%s %d", operator, &numb_arguments) == 2)
 {
#ifdef PRINTREAD
  printf("Node = `%s' arg = %d\n", operator,numb_arguments);
#endif
  op_struct = (treenode *) calloc (1, sizeof(treenode));
  strcpy(op_struct->nodename,operator);
  strcpy(op_struct->unitname,"\0");
  if (numb_arguments > 0) temp = read_exp(fp1, &(op_struct->leftchild));
   if  (numb_arguments  >  1)  temp  =  read_exp(fp1,  &(op_struct-
>rightchild));
  *expn = op_struct;
  return(INSERT_NODE);
 }
 else if (sscanf(line,"%s %s", operand, unit) == 2)
 {
#ifdef PRINTREAD
  printf("Node = %s, Unit = %s\n", operand, unit);
#endif
  opnd_struct = (treenode *)calloc (1, sizeof(treenode));
  strcpy(opnd_struct->nodename, operand);
  strcpy(opnd_struct->unitname, unit);
  *expn = opnd_struct;
  return(INSERT_NODE);
 }
 else if (sscanf(line,"%s", operand, unit) == 1)
 {
#ifdef PRINTREAD
  printf("Node = %s, Unit = unitless\n", operand);
#endif
  opnd_struct = (treenode *)calloc (1, sizeof(treenode));
  strcpy(opnd_struct->nodename, operand);
  strcpy(opnd_struct->unitname, EMPTY);
  *expn = opnd_struct;
  return(INSERT_NODE);
 }
 else {*expn = NULL;
```

```
        return(FUNNY);
       }
  }
}
tlist *load_list(fp2)
FILE *fp2;
{
 int okay = TRUE,
     temp;
 treenode *ass_stmt;
 tlist *linked_tree = NULL;
 tlist *tlist_head = NULL;
 if (fp2 == NULL)
 {
#ifdef TRACECALLS
  printf("load_list(): called with no fp2\n");
#endif
  exit(1);
 }
  while (okay)
  {
   temp = read_exp(fp2, &ass_stmt);
   switch(temp)
   {
    case END_EXPN :
      if (linked_tree)
      {
       linked_tree->next = (tlist*)calloc(1, sizeof(tlist));
       linked_tree = linked_tree->next;
       linked_tree->tree = ass_stmt;
#ifdef PRINTREAD
       printf("Above is the next link of tlist.\n\n");
#endif
      }
      else
      {
       tlist_head = (tlist*)calloc(1, sizeof(tlist));
       linked_tree = tlist_head;
       linked_tree->tree = ass_stmt;
#ifdef PRINTREAD
       printf("Above is the head of tlist\n\n");
#endif
      }
      break;
    case FUNNY :
      printf("load_list():  Invalid input.\n");
      exit(1);
      break;
    case END_OF_FP1 :
#ifdef PRINTREAD
      printf("EOF ptr reached in fp2\n\n");
#endif
      okay = FALSE;
      break;
    default       :
```

```
      break;
    } /* end switch */
   }
  return tlist_head;
}
void print_exp(root)
treenode *root;
{
 if (root==NULL) printf("NULL\n");
 printf("%s %s",root->nodename,root->unitname);
 if (root->leftchild == NULL && root->rightchild==NULL)
    printf("\n");
 else if (root->rightchild == NULL) printf(" 1\n");
 else if (root->leftchild == NULL) printf(" 1\n");
 else printf(" 2\n");
 if (root->leftchild!=NULL) print_exp(root->leftchild);
 if (root->rightchild!=NULL) print_exp(root->rightchild);
}

int match(current_list, expression)
tlist  *current_list;
treenode *expression;
{
 int found = FALSE;
 int okay;
 int compare_exp( /* exp1, exp2 */ );
 int fast_compare1( /*expression*/ );
 int fast_compare2( /*expression*/ );
#ifdef TRACECALLS
 printf("match(): called from main\n");
#endif
#ifdef PRINTMATCH
 printf("Tlist=head, do the compare\n");
#endif
 if (okay = fast_compare1(expression))
  {
   printf("expression is assign of var to var or unsigned lit\n");
   return (okay);
  }
 if (okay = fast_compare2(expression))
   {
#ifdef PRINTOKAY
      printf("okay is %d\n", okay);
#endif
      printf("expression has no units\n");
      return (TRUE);
    }
  if (okay = fast_compare3(expression))
    {
#ifdef PRINTOKAY
      printf("okay is %d\n", okay);
#endif
      printf("expression has consistant units\n");
      return (TRUE);
    }
```

```
 while (!found && current_list != NULL)
 {
#ifdef TRACECALLS
  printf("compare_exp: called from match\n");
#endif
#ifdef PRINTMATCH
  printf("Comparing %d with %d\n",current_list->tree, expression);
#endif
  found = compare_exp(current_list->tree, expression);
  if (found)
  {
    printf("valid units in expression\n");
    return(found);
  }
  else
  {
    current_list = current_list->next;
#ifdef PRINTMATCH
    printf("list != NULL check exp at next link\n");
#endif
  }
 }
 printf("invalid units in expression");
#ifdef PRINTPARSE
 printf(" ending at parsefile line %d\n",line_no);
#else
 printf("\n");
#endif
 printf("%s",start_line);
 print_exp(expression);
 printf("}\n");
 return (found);
}
int compare_exp(exp1, exp2)
treenode *exp1;
treenode *exp2;
{
 int okay = TRUE;
 if (exp1->leftchild != NULL && exp2->leftchild != NULL)
   {
#ifdef PRINTMATCH
     printf("Comparing op (%s) with op (%s)\n",exp1->nodename, exp2-
>nodename);
#endif
    if (strcmp(exp1->nodename, exp2->nodename) == 0)
      okay = compare_exp(exp1->leftchild, exp2->leftchild);
    else return(FALSE); /* operators don't match */
   }
 else if (exp1->leftchild != NULL || exp2->leftchild != NULL)
   return(FALSE);  /* different structures, so no match */
 if (okay && exp1->rightchild != NULL && exp2->rightchild != NULL)
   {
    okay = compare_exp(exp1->rightchild,exp2->rightchild);
   }
 else if (exp1->rightchild != NULL || exp2->rightchild != NULL)
```

```
   return(FALSE);  /* different structures, so no match */
 if (!okay) return FALSE;     /* save the fact that the children didn't
match */
 if (exp1->leftchild == NULL && exp1->rightchild == NULL &&
     exp2->leftchild == NULL && exp2->rightchild == NULL) {
#ifdef PRINTMATCH
     printf("Comparing (%s,%s) with (%s,%s)\n",exp1->nodename,exp1-
>unitname,
           exp2->nodename,exp2->unitname);
#endif
 if (strlen(exp1->unitname)>0 || strlen(exp2->unitname)>0)
     return (strcmp(exp1->unitname, exp2->unitname) == 0);
 else
     if (strcmp(exp1->nodename, exp2->nodename) == 0) return (TRUE);
 else return(FALSE);
     }
 return(okay);
}
int fast_compare1(expression)
treenode *expression;
{
 if (strcmp(expression->leftchild->unitname,
      expression->rightchild->unitname) == 0 &&
      expression->rightchild->rightchild == NULL &&
      expression->rightchild->leftchild == NULL)
     return (TRUE);
 else if (strcmp(expression->rightchild->nodename,
      "@@unsigned_lit")==0)
     return (TRUE);
 else return (FALSE);
}
int fast_compare2(expression)
treenode *expression;
{int result;
#ifdef PRINTCALLS
  printf("fast_compare2 called with %s as node and %s as unit\n",
      expression->nodename,expression->unitname);
#endif
  if (strcmp(expression->unitname,EMPTY)!=0)
    {
      return (FALSE);
    }
   else {
      result = TRUE;
      if (expression->leftchild != NULL)
        result = fast_compare2(expression->leftchild);
      if (result && expression->rightchild != NULL)
        result = fast_compare2(expression->rightchild);
      return (result);
    }
}
int fast_compare3(expression)
treenode *expression;
{
  if (strcmp(expression->nodename,":=")==0 &&
```

```
      expression->leftchild != NULL &&
      strcmp(expression->leftchild->unitname, EMPTY)!=0) {
#ifdef PRINTCALLS
      printf("Calling traverse with %s\n",expression->leftchild-
>unitname);
#endif
     return(traverse(expression->rightchild, expression->leftchild-
>unitname));
    }
  else return(FALSE);
}
int traverse(root, unit)
treenode *root;
char *unit;
{int result = TRUE;
#define UNSIGNED "@@unsigned_lit"
 if (root == NULL) return(TRUE);
#ifdef PRINTCALLS
 printf("traverse((%s,%s),%s)\n",root->nodename,root->unitname,unit);
#endif
 if (strcmp(root->nodename,"+")==0 || strcmp(root->nodename,"-")==0) {
   if (root->leftchild != NULL)
     if (strcmp(root->leftchild->nodename,UNSIGNED)==0) result=TRUE;
     else result = traverse(root->leftchild, unit);
   if (result && root->rightchild != NULL)
     if (strcmp(root->rightchild->nodename,UNSIGNED)==0) result=TRUE;
     else result = traverse(root->rightchild, unit);
   return(result);
   }
 else if (strcmp(root->nodename,"*")==0) {
   if (root->leftchild != NULL)
     if (strcmp(root->leftchild->nodename,UNSIGNED)==0)
        if (root->rightchild != NULL)
          return (traverse(root->rightchild,unit));
        else return(FALSE); /* unary * not allowed */
     else if (root->rightchild != NULL)
        if (strcmp(root->rightchild->nodename,UNSIGNED)==0)
          return (traverse(root->leftchild,unit));
        else return(FALSE); /* multiplcation by non unsigned_lit */
     else return(FALSE); /* unary * not allowed */
   else return(FALSE); /* unary or leaf * not allowed */
   }
 else if (strcmp(root->nodename,"/")==0) {
   if (root->leftchild != NULL)
     if (root->rightchild != NULL)
        if (strcmp(root->rightchild->nodename,UNSIGNED)==0)
          return (traverse(root->leftchild,unit));
        else return (FALSE); /* division by non unsigned_lit */
      else return (FALSE); /* unary / not allowed */
    else return (FALSE); /* unary or leaf / not allowed */
    }
 else if (strcmp(root->unitname,unit)!=0) return(FALSE);
 else return(TRUE);
}
```

APPENDIX C

SAMPLE PAGE FROM RULEBASE

```
{ 1257
:= 2
Intermediate kilometersseconds
* 2
/ 2
SquadLoc[].Endur newtons
Army[][].Endurance[] newtons
Army[][].V0[] kilometersseconds
}
{ 1333
:= 2
A1 perkilometers
* 2
/ 2
@@unsigned_lit
* 2
Params.XDelta kilometers
Params.YDelta kilometers
+ 2
- 2
- 2
Terrain[] kilometers
Terrain[] kilometers
Terrain[] kilometers
Terrain[] kilometers
}
{ 1337
:= 2
A2
* 2
/ 2
@@unsigned_lit
Params.XDelta kilometers
- 2
* 2
N
- 2
Terrain[] kilometers
Terrain[] kilometers
* 2
+ 2
N
@@unsigned_lit
- 2
Terrain[] kilometers
Terrain[] kilometers
}
```

APPENDIX D

SAMPLE PAGE FROM OUTPUT

```
{ 370
:=  2
Arg
/  2
Pos.X kilometers
Params.XDelta kilometers
}
{ 371
:=  2
Arg
/  2
Pos.Y kilometers
Params.YDelta kilometers
}
{ 447
:=  2
Effect
/  2
-  2
Army[][].MaxSlope
/  2
-  2
Alt kilometers
Alt kilometers
MoveDist kilometers
Army[][].MaxSlope
}
{ 500
:=  2
Arg
/  2
Pos.X kilometers
Params.XDelta kilometers
}
{ 501
:=  2
Arg
/  2
Pos.Y kilometers
Params.YDelta kilometers
}
```

LIST OF REFERENCES

Beizer, B., *Software Testing Techniques*, Van Nostrand Reinhold, 1990.

Bhargava, H. K., *Dimensional Analysis in Mathematical Modeling Systems, A Simple Numerical Method,* Naval Postgraduate School, Monterey, California, February 1991.

Dobieski, A. W., "Modeling Tactical Military Operations", *Quest*, pp.1-25, Spring 1979.

"Glossary of Software Enfinnerring Terminology," ANSI-IEEE Standard 729-1983, 1983.

Howden, W. E., "A Survey of Static Analysis Methods," *Tutorial: Software Testing and Validation Techiques,* New York: IEEE Press, pp. 101-115, 1981.

Howden, W. E., "A Survey of Dynamic Analysis Methods," *Tutorial: Software Testing and Validation Techiques,* New York: IEEE Press, pp. 210-230, 1981.

Jensen, K. and Wirth, N., PASCAL User Manual and Report, Springer-Verlang, 1974.

Karr, M. and Lovemen, D.B., "Incorporation of Units inot Programming Languages, " *Communications of the ACM*, vol. 21, no. 5, pp. 385-391, May 1978.

Mason,T . and Brown, D., *lex and yacc*, O'Reilly & Associates, Inc., 1991.

Shimeall, Timothy, "Conflict Specification," Naval Postgraduate School, Monterey, California, 1990.

Shimeall, Timothy and Leveson Nancy, "An Empirical Comparison of Software Fault Tolerance and Fault Elimination," IEEE Transaction on Software Engineering, February 1991.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 52 Naval Postgraduate School Monterey, California 93943-5002	2
3.	CPT Judy A. Browning 96 Ravenwood Way Warner Robins, GA 31093	4
4.	Timothy Shimeall Code CS/Sm Naval Postgraduate School Monterey, California 93943-5100	4
5.	Amr Zaky Code CS/Za Naval Postgraduate School Monterey, California 93943-5100	1
6.	Robert B. McGhee Code CS/Mz Naval Postgraduate School Monterey, California 93943-5100	1
7.	CDR Thomas J. Hoskins Code 37 Naval Postgraduate School Monterey, California 93943-5100	1

Thesis
B82401 Browning
c.1 An empirical study of
fault detection by static
units-consistency analy-
sis.